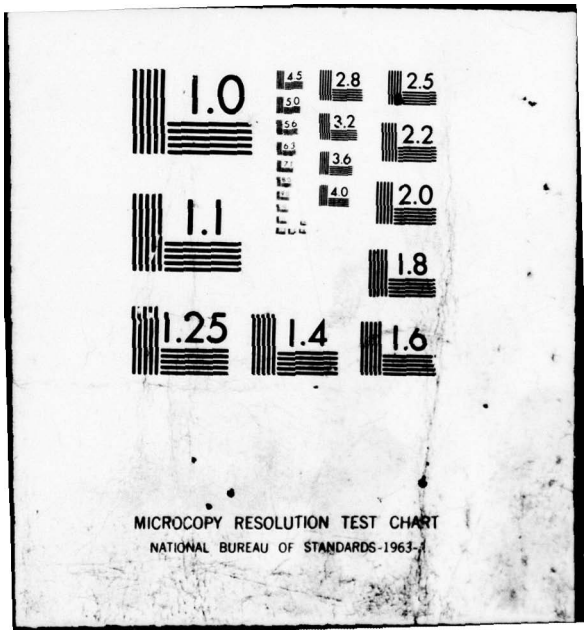
AD-A073 173 MITRE CORP BEDFORD MA F/G 9/2
A KERNEL-BASED SECURE UNIX DESIGN.(U)
MAY 79 J P WOODWARD, G A NIBALDI F19628-78-C-0001
UNCLASSIFIED MTR-3499 ESD-TR-79-134 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ESD-TR-79-134

LEVEL II 12

MTR-3499

A KERNEL-BASED SECURE UNIX DESIGN

BY J. P. L. WOODWARD AND G. H. NIBALDI

MAY 1979

Prepared for

DEPUTY FOR TECHNICAL OPERATIONS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts

Project No. 572N, 8010
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract No. F19628-78-C-0001

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

Daniel R. Baker

DANIEL R. BAKER, Captain, USAF
Technology Applications Division

Charles J. Grewe, Jr.

CHARLES J. GREWE, JR. Lt Colonel, USAF
Chief, Technology Applications Division
Directorate of Computer Systems Engineering

Normand Michaud

NORMAND MICHAUD, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Technical Operations

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) 1. REPORT NUMBER: ESD-TR-79-134

(9) 2. GOVT ACCESSION NO.: Technical rept.

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE (and Subtitle): A KERNEL-BASED SECURE UNIX DESIGN

5. TYPE OF REPORT & PERIOD COVERED

(14) 6. PERFORMING ORG. REPORT NUMBER: MTR-3499

(10) 7. AUTHOR(s): J. P. L. WOODWARD, G. A. NIBALDI

(15) 8. CONTRACT OR GRANT NUMBER(s): F19628-78-C-0001

9. PERFORMING ORGANIZATION NAME AND ADDRESS
The MITRE Corporation
P. O. Box 208
Bedford, MA 01730

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Project No. 572N, 8010

11. CONTROLLING OFFICE NAME AND ADDRESS
Deputy for Technical Operations
Electronic Systems Division, AFSC
Hanscom AFB, MA 01731

(11) 12. REPORT DATE: MAY 1979

13. NUMBER OF PAGES: 94

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

(12) 95p.

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
COMPUTER SECURITY
SECURITY KERNEL
UNIX

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The UNIX Time-Sharing System was developed by Bell Laboratories for the DEC PDP-11 series of computers. The Secure UNIX Prototype Project has addressed the problem of defining a secure version of UNIX that preserves as many as possible of its desirable qualities: completeness and simplicity of design, low cost, and an extensive amount of quality software. This report presents a prototype design for a Secure UNIX system based on security kernel technology previously developed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

235 050

ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project Nos. 572N, 8010. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

The authors would like to thank K. J. Biba for the original Secure UNIX design from which this design has evolved.

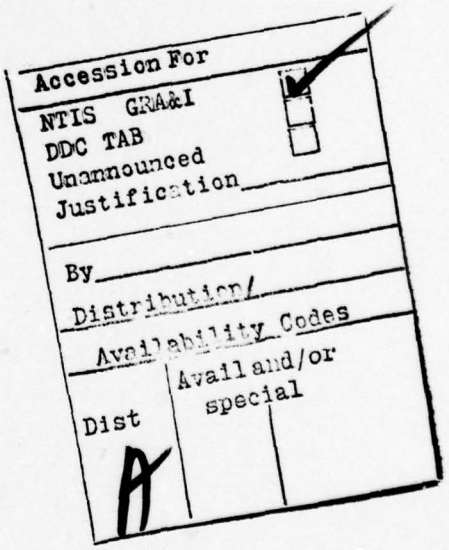


TABLE OF CONTENTS

		Page
LIST OF ILLUSTRATIONS		5
SECTION I	INTRODUCTION	6
	PURPOSE	6
	PROJECT GOALS	6
	PLAN OF PAPER	7
SECTION II	SECURE UNIX DESIGN CONSIDERATIONS	8
	UNIX ATTRIBUTES	8
	UNIX Object Structure	8
	UNIX Operations	11
	UNIX Protection Policy	12
	UNIX Deficiencies	13
	SECURE UNIX PROTECTION POLICY	14
	SECURE SYSTEM DESIGN	14
	SECURE UNIX ARCHITECTURE	15
SECTION III	SECURE UNIX KERNEL ARCHITECTURE	18
	DESIRED PROPERTIES	18
	Kernel Objects	18
	KERNEL FACILITIES	21
	General Properties	21
	Kernel Protection Policy	22
	Kernel Object Structure	24
	Kernel Operations	28
	KERNEL INTERNAL ARCHITECTURE	40
	Abstract Machine Decomposition	40
	Process Decomposition	54
SECTION IV	PRIVILEGED SUBSYSTEMS	62
	PROTECTION ISSUES	62
	ROOT PROCESS	63
	DISCRETIONARY AUTHENTICATOR	64
	PORT MANAGER	64

TABLE OF CONTENTS (Concluded)

		Page
SECTION V	THE SECURE UNIX EMULATOR ARCHITECTURE	67
	PROTECTION POLICY	67
	OBJECT STRUCTURE	68
	Information Processors: Process Family	68
	Information Storage and I/O: File System	69
	OPERATIONS	72
	Process Operations	73
	File Operations	76
	EMULATOR INTERNAL ARCHITECTURE	79
	Data Bases	79
	Emulator Subsystems	82
SECTION VI	SUMMARY	88
	ACCOMPLISHMENTS	88
	DISAPPOINTMENTS	90
	HARD PROBLEMS	91

LIST OF ILLUSTRATIONS

Figure Number		Page
1	UNIX File Structure	9
2	SUNIX Architecture	16
3	Kernel Object Attributes	29
4	Kernel Operations	39
5	Kernel Abstract Machines	41
6	Non-Terminal Device Manager Process Structure	56
7	Terminal Device Manager Process Structure	57
8	Segment Manager Architecture	59
9	User Process Manager Process Structure	60

SECTION I

INTRODUCTION

PURPOSE

UNIX is a time-shared, interactive, multiprogrammed operating system for the DEC PDP-11 series of computers. Its qualities of completeness and simplicity of design, low cost, and an extensive amount of quality software have made it attractive to a large community of users to which such computing power was never before available. These same properties make it attractive as a candidate for a secure operating system.

The DoD community has many of the same computational needs as the general computing community. These needs are compounded by a need to protect classified information. Thus a Secure UNIX system would address many of these problems in a powerful, cost-effective manner. This paper will attempt to address the issues in the development of a Secure UNIX through the presentation of a prototype design for such a system.

PROJECT GOALS

The Secure UNIX Protoype Project at MITRE, sponsored by both ESD and DARPA, is addressing the problem of defining a secure version of the UNIX operating system that preserves as many as possible of the desirable qualities of the UNIX system. In particular, we have had the following goals during this project:

a. incorporation of DoD security policy;

b. preservation of performance;

c. transparency of operation to unilevel user programs;

d. minimal verified mechanism to support the policy.

The imposition of these goals on UNIX has caused the sacrifice of a primary UNIX attribute (attractive to many): the extreme malleability of UNIX to be molded into UNIX mutations deemed more useful for one application or another. A verified system cannot provide the same degree of flexibility that UNIX itself provides. However, this loss is offset by the availability of a secure operating system.

PLAN OF PAPER

The paper is organized into five additional sections. The next section contains a discussion of the general architecture and properties proposed and constructed for a Secure UNIX. Section III decomposes the general architecture of Section II to discuss the detailed architecture of the Security Kernel portion of Secure UNIX. Section IV describes the Secure UNIX privileged subsystems, processes that must circumvent ordinary security controls to perform certain necessary functions. Section V builds on the Kernel base of Section III to define the set of non-verified software that provides the "illusion" of UNIX to users of the system: this set of software is termed the Secure UNIX Emulator since it emulates the UNIX interface to user programs. The final section concludes the paper with a brief resume of some of the successes and failures of the prototype.

SECTION II

SECURE UNIX DESIGN CONSIDERATIONS

This section reviews the issues considered in the design of Secure UNIX (SUNIX). First the UNIX attributes that most directly affect the SUNIX design are presented. Next the protection policy chosen for SUNIX is stated, and finally the general architecture of SUNIX is described.

UNIX ATTRIBUTES

UNIX provides a general interactive environment. It draws much of its technology from a distinguished group of predecessor systems, including MULTICS and TENEX. It provides device independent input/output, a hierarchical file naming system, user level multiprogramming, and a sophisticated set of applications programs. A full description of the UNIX system can be found in [1] and [2].

This discussion of UNIX attributes is centered around a discussion of the abstract objects (and operations on these objects) that UNIX provides, for it is these attributes of UNIX that most directly affect the design of SUNIX.

UNIX Object Structure

UNIX defines two basic types of objects: file systems (which include input/output) and processes. Their attributes are summarized below.

A file system in UNIX is a large block of mass storage such as random access tapes, disks, or logically defined portions thereof. A UNIX system may have an arbitrary number of file systems in use at one time. File systems other than the "root" file system may be "mounted" and "unmounted" at any time.

Each filesystem is partitioned into a linearly addressed set of 512 byte blocks. Some blocks are used to organize the rest and consist of blocks that define the free list of available blocks, and, more importantly, the table of i-nodes defined for this file system. Each i-node describes a file in the file system. The UNIX file structure is diagrammed in figure 1.

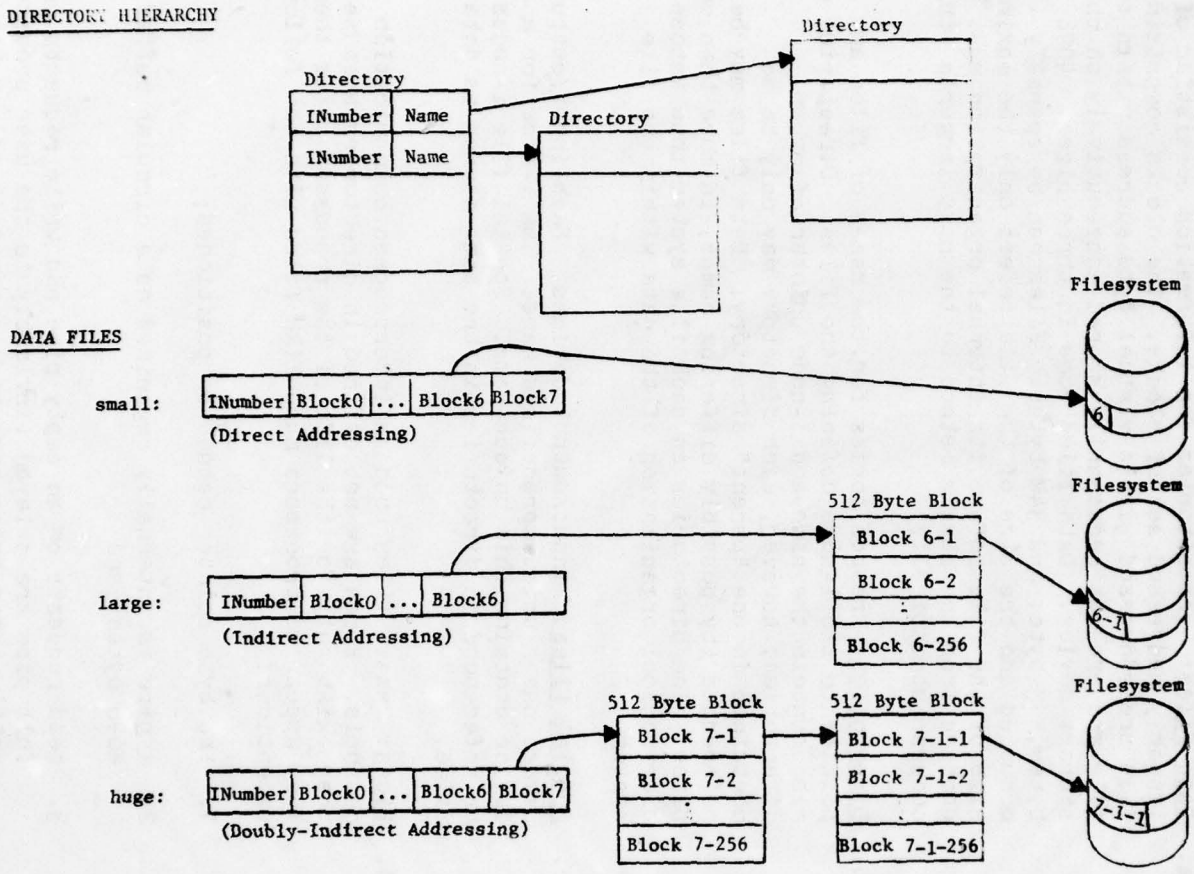


Figure 1. UNIX File Structure

A file system is organized into files of various types, each of which is described by an i-node. I-nodes are of the following types:

a. Data files: repositories for information consisting of a linearly addressed set of blocks. The blocks comprising a file are addressed with a virtual byte address within the file and are not necessarily stored consecutively on the storage device. Data files come in three sizes: 4096 bytes, 1M byte, and 34M bytes. Files can be sparsely occupied and the size of the file refers only the maximum size of the file before its internal organization must be changed from one size's method to the next larger's (this occurs automatically).

b. Directories: respositories for the names of files and a pointer to the i-node defining the files. Directories may also contain the name and i-node pointer of other directories; however, each directory may only be so contained in one "parent" directory. Data files may be referenced (by possibly differing names) in more than one place. The directories in each file system thus impose a hierarchical organization of the data within the file system.

c. Special files: input/output devices. Each input/output device has a unique predefined name. The i-node for a device contains this unique name. Special file i-nodes may be referenced in directories in the same manner as data files.

d. Pipes: small files used as interprocess communication channels. Pipes are not defined in directories because they exist only for the life of the processes using them. They appear to processes much like files with the following exceptions:

    1. the byte pointer cannot be positioned;

    2. a pipe is internally organized as a circular buffer of 4096 bytes; and

    3. read requests on an empty pipe and write requests on a full pipe are delayed (invisibly to the user process) until either bytes have been entered (read) or bytes have been removed (write).

The other basic object that UNIX supports is processes. A process is created for each user when he logs in, and a user may create (fork) an arbitrary number of processes during a session. Processes have the following attributes:

a. a name;

b. a virtual address space composed of 128k bytes: 64k bytes of instructions and 64k bytes of data. The virtual address space is composed of three segments: the read-only (possibly) shared text (instructions), read-write data (and instructions), and read-write stack;

c. a set of currently accessible files, pipes, or special files (about 15); and

d. a set of timers and intraprocess traps (interrupt, "quit", memory fault, etc.).

UNIX Operations

UNIX files can be created and deleted by any process. The creator of a file becomes the owner of the file; only the superuser can change the owner of a file. The owner or superuser can change the protection attributes of the file. Contiguous byte streams can be read or written from and to files, pipes, or special files. Files, pipes, and special files can be added to or removed from the set of currently accessible files for each process (up to a maximum of 15).

UNIX processes can spawn (fork) other processes that initially possess an exact duplicate of the address space of their parent. Terminal devices and other processes can transmit preemptive (interrupt-like) signals to other processes. Processes can construct interprocess communication channels among descendants of a common ancestor via pipes: a circular buffer looking (to the processes) like a peculiar type of file. Processes may create, delete, read, write, and position the byte pointer of files. Processes may also alter the protection attributes of files that they own, i.e., when the user id of the process is identical to that of the file.

Processes may set a "virtual" timer for themselves, allowing a timeout signal at the end of process specified period. Processes are scheduled via an adaptive algorithm given priority to interactive processes (viz., processes having many short interactions with the user at his or her terminal). Processes that are "compute-bound", i.e., have little or no terminal interaction,

are given low-priority for access to cpu and core resources. Low priority processes are given less frequent access to resources but are given a larger quantum of time in which to make use of them. High priority processes, on the other hand, receive frequent quanta of resource, but a smaller quantum.

UNIX Protection Policy

UNIX provides a protection policy designed to protect access to each individual file, directory, and special file in a file system. UNIX protection is currently based on the notion of "users" and "groups", and is discretionary in nature. Each UNIX object (files, directories, special files, and processes) belongs at any point in time to a "user" and a "group". The set of users and the set of groups are each drawn from the set of integers from 0 to 255. One particular user, user zero, is termed the "superuser"; this user can violate all access controls.

Each file and directory contains three items of protection information:

a. the user owning this object;

b. the group owning this object; and

c. the access permissions for processes

    1. belonging to the same user;

    2. belonging to the same group but a different user; and

    3. every other process.

The access modes are a subset of the set {read, write, execute} for files, and {read, modify, and search} for directories. Either the owning user or the superuser may alter the access modes. Only the superuser can alter the user/group of a file/directory.

Processes inherit the same user/group across a fork operation. A process belonging to the superuser may alter its user/group to any other user/group. No other process may alter its user or group except thru the "set_user" or "set_group" mode of program execution. An owning user may designate that a specified program file possess the attribute of the "set_user" or "set_group" mode. In this instance, a process belonging to another user or group, if it has execute permission, will temporarily (during the time the specified program file is executed by the process) assume the user/group access privileges of the user/group that owns the program file.

This mechanism provides a limited form of domain protection for proprietary programs or data.

UNIX Deficiencies

Despite its advantages, UNIX has a number of deficiencies as a general purpose computer utility. These deficiencies are presented because they have been addressed in the design of SUNIX.

First, UNIX lacks a true "virtual memory". That is, processes may not concurrently share named storage within their address space. Some research is proceeding along these lines; however the "standard" UNIX system is sadly deficient in this area. This deficiency makes more difficult the task of implementing sophisticated data management systems allowing concurrent access to a data base by many users.

This problem is compounded by the lack of good interprocess communication mechanisms. Only two mechanisms are currently provided:

a. the interprocess signal mechanism - no data can be transmitted over this mechanism; and

b. the pipe facility - pipes are only in existence during the lifetimes of the processes using/creating the pipes. Pipes cannot be shared among processes that have no common ancestor and pipes are not a viable mechanism from which to provide an interprocess coordination mechanism for shared data bases.

Both mechanisms, as noted above, are inadequate for use in the construction of general purpose interprocess communication/coordination mechanisms necessary for current architectures involving multiple processes accessing shared data bases. The current UNIX mechanisms are more than adequate for the transmission of data between sequential phases of an algorithm.

A more serious problem is one of inconsistent abstraction. That is, certain "abstract" facilities provided by UNIX, directories for instance, are not completely "hidden" at the system interface. In particular, the internal structure of directories is not only visible to programs executing outside the "hard-core" of the UNIX system; this information is required for many necessary functions of the system. For instance, the program that lists the contents of directories relies on its ability to read the directory as a data file in order to determine what files are defined within it. A more serious instance of the same problem is the "visibility" of the

internals of the UNIX filesystems. A UNIX filesystem (as defined above) consists of some set of directories, files, and special files organized (mapped) on the physical address space of some device. UNIX currently allows, and indeed requires for some functions, the ability to read or write the special file corresponding to the filesystem. Thus, the access controls and data organization imposed by the filesystem structure can be defeated by access to the file via its corresponding special file. Both these "facilities" present special problems in the design of a Secure UNIX.

The above issues suggest two tenets that have directed the design of SUNIX. First, to be useful in a large set of the projected uses of SUNIX, it must provide the mechanisms for IPC and virtual memory needed by those applications. Second, some of the historical "oversights" of UNIX can and should be modified and "fixed" in SUNIX.

SECURE UNIX PROTECTION POLICY

SUNIX must support both discretionary and non-discretionary access controls. The non-discretionary controls SUNIX will support is the partially ordered set of security levels and categories defined for military computer systems [3,4]. SUNIX will support 16 security levels and 64 categories.

The discretionary controls that SUNIX supports are identical to those provided by UNIX, for compatibility reasons.

SECURE SYSTEM DESIGN

The design of SUNIX has been based on our background in secure multilevel computer systems. Secure systems began from a requirement for secure sharing of information at a variety of security levels. Analysis quickly revealed two problems:

a. define a security policy constraining access of information processors to information repositories; and

b. guarantee that the policy is ALWAYS enforced.

The definition of security policy turned out to be more of a task than expected, yet the military already had much of the policy formulated. The real task became the guarantee of enforcement.

The essence of the task became the following: to guarantee security there must exist a rigorous verification of the security

properties of security sensitive programs. Since such verification is expensive, time-consuming, and seemingly only possible for small programs, the verification, and thus the security sensitive programs, should be confined to a small part of the system. These programs should create an environment for the other programs such that the remaining programs cannot violate the security policy. Thus the remaining programs need not be rigorously verified (at least with respect to security). The security sensitive programs are termed the security kernel.

The architecture for secure systems is thus derived from the notion of the security kernel. All security kernels designed to date operate on the basis of certain assumptions. These assumptions are:

a. the kernel provides a process structured environment;

b. each process (supported by the kernel) has available to it a set of objects that can be accessed only by using defined operations;

c. each process is confined with respect to the defined protection policy;

d. each process can execute whatever programs it pleases: because it is confined it cannot, by construction, violate security; and

e. within each process there may exist several synchronous domains of access privilege recursively defining new sets of objects from the object structure supplied by the security kernel.

What is important to note is that for any security kernel of some sophistication, the security kernel not only controls access to objects, it must define the objects and construct operations to access these defined objects.

SECURE UNIX ARCHITECTURE

The architecture for SUNIX is security kernel based and is diagrammed in figure 2.

At the base of SUNIX is a security kernel. The security kernel defines a process structured environment wherein each process possesses access permission based on its security level. In

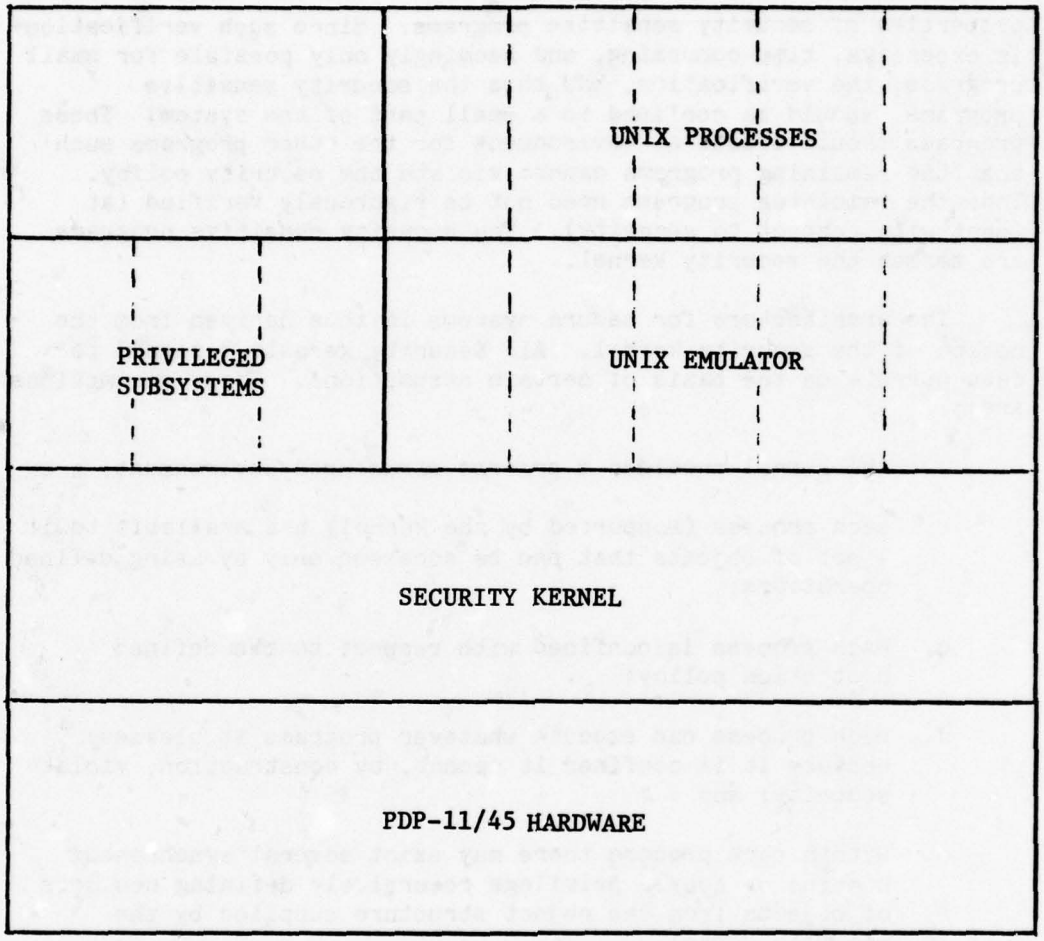


Figure 2. SUNIX Architecture

particular, the security kernel defines three types of protected objects:

a. segments: the unit of information storage;

b. devices: the process-like mechanism for input/output; and

c. processes: the information processors within the system.

Each process may map segments into its virtual memory to gain access to the information contained within them.

The memory space of each process is partitioned into two ordered domains: the supervisor and the user, each of equal size. The supervisor domain is the controlling domain. It makes all requests for kernel service on kernel defined objects and interprets requests from the user domain for service. In SUNIX the supervisor domain of each kernel process that corresponds to a UNIX process shall execute a reentrant program termed the UNIX Emulator that provides the "illusion" of UNIX to programs executing in the user domain.

From the above discussion it should be apparent that the intended architecture for SUNIX will provide a direct correspondence between kernel supported processes and UNIX processes. Each SUNIX process, as indicated above, will have two domains of execution outside the kernel. The first domain, the supervisor, contains the necessary programs and data bases (contained in segments) to construct a UNIX-like environment from kernel provided objects and operations. The second domain, the user domain, will appear much like the process environment currently supported by UNIX. Current programs should, if they do not require the capability to access and modify information at several security levels, execute transparently.

SECTION III

SECURE UNIX KERNEL ARCHITECTURE

This section is devoted to a discussion of the SUNIX kernel architecture. First, general architectural principles and desired kernel properties will be discussed. Next, the kernel interface, i.e., the interface that non-verified, untrusted programs may use is presented. Finally, the internal architecture of the kernel is discussed, with primary emphasis on the decomposition into "levels of abstract machines" and the orthogonal decomposition into internal kernel processes.

DESIRED PROPERTIES

An operating system requires the properties of efficiency of resource usage, correctness (reliability) of its function, simplicity and completeness of its operations, and minimal size. A security kernel is no different in these characteristics. Where it differs is in a strengthening of these requirements and in the addition of a selected protection policy. The following paragraghs will address desired functional qualities of the kernel interface and the necessary protection policies.

Kernel Objects

The kernel, by its very nature, must provide a number of abstract objects which can be manipulated through primitives at the kernel interface. There are three kinds of objects to consider: processes, segments, and input/output devices.

Processes

Processes represent the basic computational element of the kernel-provided computing system. Current application system architectures emphasize the decomposition of programs into a family of interacting, logically asynchronous processes. Experience with data base architectures indicates that processes should possess some form of virtual memory access to information storage. These considerations suggest the following process qualities:

a. processes should be able to create (spawn) other processes;

b. processes should be able to send and receive information from other processes (InterProcess Communication, or IPC);

c. processes should be able to access a virtual address space; and

d. the process virtual environment should contain several domains of access privilege, thus permitting construction of "supervisors" within processes.

The IPC mechanism should have several important characteristics. The first of these is the structure of the IPC message. The IPC message should essentially contain two fields: the name of the sending process (unforgeably supplied by the kernel), and a data field supplied by the sending process. It would be quite useful to have a data field of arbitrary length; however, simplicity and kernel minimality considerations argue against it. A better solution is to design the data field size and the segment naming structure so as to permit the transmission of a segment name within the data field of the IPC message. This structure permits the construction (and transmission) of arbitrary sized messages through the use of virtual memory.

It should be noted that the combination of the facilities for message-based IPC, virtual memory, and an instruction set that supports indivisible READ-TEST-THEN-MODIFY operations on virtual memory permits the construction of interprocess locks on data bases defined externally to the kernel.

The process creation mechanism should have two basic qualities: it should allow the creating process to specify some part of the environment of the new process so as to allow for easy bootstrapping within that new process, and processes should be capable of spawning processes of greater or equal security level.

The virtual memory of a process should allow for the mapping of information segments to some partition of each process's virtual memory. The process should not be aware of the location of the segment (primary or secondary memory).

Segments

Segments constitute the basic information repository within the system. Segments have a name, protection labels, a size, and an information content. Two issues are important regarding segments. First, the name of a segment should be capable of being passed as data via an IPC message. This requirement argues for a fixed length name of the unique identifier variety. Under this scheme each segment name is guaranteed to be unique and unmodulatable. A convenient mechanism to accomplish this is by using the value of a system calender clock, with a resolution substantially less than the

rate of segment creation requests. The second issue regards the size of segments. A large portion of the projected uses for secure minicomputer systems, including SUNIX, fall into two categories: network processors and text management. Informal studies both of average paragraph length and of packet size suggest that a segment size of about 512 bytes is desirable. Other considerations, such as the size of program space and temporary virtual memory storage (buffers and stacks) suggest a larger size ranging from 1k to 8k bytes (particularly for PDP-11-like architectures). Thus a variety of segment sizes is required. This requirement for a variety of segment sizes must be balanced with the requirements of kernel simplicity, particularly in the area of resource management.

Processes should have the capability to create segments of any size and delete any segment that they can name. The only enforced constraints are protection ones. In particular, for mandatory security, processes should only be capable of creating and deleting segments at a greater or equal security level. However, no information as to the success or failure of the operation can be passed to the process if the segment operated upon has a security level strictly greater than the security level of the process.

Devices

Devices can be regarded as special purpose processes. They should execute asynchronously. They should (if possible) present a uniform interface. Both character-at-a-time and DMA devices should be supported. Support for the modification of the current (working) security level of each device should be provided. In the case of interactive terminals, an easy-to-use mechanism/protocol for terminal (user) initiated security level change (as opposed to process initiated security level change) should be provided.

An issue concerning devices that should be examined is the extent to which device "drivers" are embedded within the kernel. The system overhead in invoking the kernel is generally substantial. An even larger overhead is the time needed to switch between user (non-kernel) processes. Thus, both the number of process switches and the number of kernel invocations should be minimized. This thought has particular impact on the kernel interface to character-at-time terminals. If the kernel allows only one character to be output at a time, at least one kernel call and two process switches are required per character input or output. In some architectures this overhead could increase to four or more process switches.

The above efficiency concern would suggest that the kernel support some form of byte stream input/output to these devices, thus embedding some buffering capability within the kernel. Buffering

goes counter to the conventional wisdom of a "simple", non-interruptable kernel. A kernel would then have to deal with the issue of handling each character's interrupt. However, the mechanism that is designed to "simply" handle these character interrupts within the kernel can also be used to efficiently implement some other asynchronous facilities within the kernel.

Trusted Processes

An added kernel "feature" is the capability to designate certain processes as "trusted". That is, these processes have the ability to violate certain of the access controls enforced for processes in general. This capability imposes the added burden of some access control for the protection of these "trusted" processes from sabotage: improper modification of the data/instructions used by them. To address this problem a kernel should have an added protection policy: an integrity policy.

KERNEL FACILITIES

General Properties

The SUNIX kernel provides, as discussed above, three basic types of objects: processes, segments, and input/output devices. Processes are the active computational element provided by the kernel, so it is the processes that make requests to the kernel. The kernel provides processes with the following general capabilities:

a. the capability to create and delete processes;

b. the capability to send/receive fixed length messages between processes;

c. the capability to request input/output from asynchronous devices and to receive a completion notification;

d. the capability (for suitably privileged processes) to alter the current protection label for a device (security level is a special case);

e. the capability to create and delete several sizes of segment; and

f. the capability to map segments into portions of a process`s virtual memory and access the segment as memory.

Kernel Protection Policy

Three protection policies are supported by the SUNIX security kernel:

a. the mandatory security policy;

b. the mandatory integrity policy; and

c. a version of the UNIX discretionary protection mechanism.

Mandatory Security Policy

The mandatory security policy defines the notion of DoD security policy [3,4] within SUNIX. It is intended to address the problem of prevention of compromise of classified information. It is represented by the following elements:

a. a security level drawn from the set of security levels defined by the cross-product of a set of sixteen (16) security classifications and the powerset of a set of sixty-four (64) security categories;

b. a set of information processors, termed subjects, to each of which is assigned a security level;

c. a set of information repositories, termed objects, to each of which is assigned a security level;

d. the simple security condition [5], enforced for all operations of subjects upon objects, that requires a subject to have a security level greater than or equal to the security level of any object that is observed by the subject; and

e. the security *-property [5], enforced for all operations of subjects upon objects, that requires a subject to have a security level less than or equal to the security level of any object that is modified by the subject.

Mandatory Integrity Policy

The mandatory integrity policy complements the mandatory security policy by defining a set of access controls against information sabotage. The policy is defined by the following elements:

a. a set of integrity levels drawn from the cross-product of a set of integrity classifications (of cardinality sixteen (16)) and a set of eight (8) integrity categories;

b. a set of subjects - the same set as for the mandatory security policy;

c. a set of objects - the same set as for the mandatory security policy;

d. the simple integrity condition [6], enforced for all operations of subjects upon objects, that requires the integrity level of a subject to be greater than or equal to the integrity level of an object if the subject modifies the object; and

e. the integrity *-property [6], enforced for all operations of subjects upon objects, that requires the integrity level of a subject to be less than or equal to the integrity level of an object if the object is observed by the subject.

Discretionary Policy

A UNIX-like discretionary mechanism is provided by the kernel to support the UNIX protection mechanism on files, directories, and special files. This policy differs from the preceding policies in that these discretionary protection attributes can be altered, for existing objects, by an appropriately privileged subject. The elements of protection mechanism include:

a. a set of user identifiers drawn from the integers 0 to 255;

b. a set of group identifiers drawn from the integers 0 to 255;

c. a set of subjects to which is assigned both a user and group identifier at subject creation time;

d. a set of objects to which is assigned both a user and group identifier (sometimes termed "owner") either at object creation time or by an appropriately privileged subject;

e. a set of access modes including NULL, READ-ONLY, and READ-WRITE;

f. a set of allowed access permissions for each object, one set for each of the following sets of subjects:

1. those subjects possessing differing user and group identifiers;

2. those subjects possessing differing user identifiers but the same group identifier; and

3. those subjects possessing the same user and group identifiers.

The last element defines the class of access control policies that can be represented by this mechanism.

Process Privilege

The final protection policy notion to be addressed is the concept of "subject" (in particular, process) privilege. The above qualification is made because, while in general the notion of privilege can apply to all subjects within a system, in practice it is restricted to that class of subjects termed processes. Devices, the other major class of subjects, are considered to be always unprivileged.

Process privilege is the ability to violate one or more of the above five access control policies:

a. the simple security condition;

b. the security *-property;

c. the simple integrity condition;

d. the integrity *-property; and

e. the discretionary protection mechanism.

Processes possess an attribute, defined at process creation time, that is termed process privilege, and that defines which of these access controls that process may violate. A process having a non-null set of privileges is termed privileged. Privileged processes are used in SUNIX to implement Trusted Subjects [7]. A privileged process may create processes having only a subset of its privilege.

Kernel Object Structure

The structure of the three types of kernel objects is outlined below.

Information Storage: Segments

Segments are the primary information repositories supported by the kernel. They possess the following attributes.

a. A name: a 48-bit unique integer, obtained from the system time-of-day clock, that uniquely labels each segment during the life of the system. The high-order eight bits of this name denote a filesystem, and the low-order forty bits uniquely identify the segment within the filesystem. The concept of a kernel filesystem is somewhat similar to the concept of a UNIX filesystem. A kernel filesystem corresponds directly with a physical storage device: a spindle of disk, a DECtape drive, or any logical portion thereof.

b. An access level.

c. A domain: Three domains of segment are defined: kernel, supervisor, and user. Kernel segments can only be mapped into the kernel domain virtual memory of any process, supervisor segments may only be mapped into the kernel or supervisor domain virtual memory, and user segments may be mapped anywhere.

d. User and group identifiers.

e. Discretionary permissions.

f. A size: The kernel supports multiple sizes of segment to accommodate a variety of applications. Segments can be had in sizes of two to the n bytes for n between nine and thirteen (512, 1024, 2048, 4096, and 8192 bytes).

g. A value: The contents of the segment.

Input/Output: Devices

Devices are special purpose processes, having a limited instruction set, whose function is to provide an interface to the kernel external environment. Devices have the following attributes.

a. A name: A device name is a 32 bit unique integer specified at the time the system is created.

b. An access level: Each device possesses a "current" value of its access level. This value must always be less than or equal to its "maximum" access level.

c. A "maximum" access level: The high water mark of the device, specified at system compile time.

d. A domain: Always implicitly supervisor. Only supervisor mode programs can directly access devices.

e. A "current" process: The name of the process currently associated with this device. This field is here to allow direction of terminal signals to the appropriate process.

f. User and group identifiers.

g. Discretionary permissions.

h. A value: A device value can essentially be considered an array (possibly very large) of bytes.

i. A byte pointer: An index into the value of the device indicating the "current" position of the device. Some devices permit arbitrary positioning of this pointer (block addressable devices) while some allow only limited positioning (tapes permit rewind) and some devices permit no positioning (terminals).

j. A request queue: A FIFO list of process requests to the device to perform a service. Services are of three types:

   1. an input request for transfer of a byte string into virtual memory beginning from the current position of the byte pointer;

   2. an output request for transfer of a byte string from a specified portion of virtual memory (some segment) to the device beginning at the current position of the byte pointer - for both the input and output requests the position of the byte pointer is updated; and

   3. a non-data transfer request - such as a positioning request.

Information Processors: Processes

Processes are the primary information processing element supported by the kernel. Processes have the following attributes:

a. a name, drawn from the same space as device names, however, derived from the system clock with the guarantee that no two processes concurrently defined have the same name;

b. an access level defined at the time the process is created and unchangeable during the life of the process;

c. a "current" domain: each process possesses three domains of execution - kernel, supervisor, and user. Only the supervisor and user domains are accessible to non-kernel programs.

d. a "previous" domain: the previous domain of a process is never "greater than" the "current" domain, and is used to direct intra-domain transfer instructions of the hardware;

e. a user and group identifier: specified at the time of process creation and unchangeable during the life of the process;

f. a privilege type: denoting whether the process is privileged and if so, which access controls the process can violate;

g. miscellaneous hardware context: indicators, general registers, program counter, and stack pointer;

h. an integer valued priority: this value controls (to some extent) the priority and frequency with which this process can gain access to the processor for computation;

i. an interprocess message queue: a FIFO list of 16 byte messages sent to the process from other processes and input/output devices; Note that the kernel provides internal queuing of messages. Message buffer exhaustion will cause the kernel to prematurely abort;

j. an inter and intraprocess trap vector: a Boolean vector containing notification of the existence of a preemptive trap condition forcing kernel vectoring of a trap condition through the supervisor entry point (within each process) for traps;

k. a virtual interval timer: the timer is settable by the supervisor domain of each process and will generate an intraprocess trap when it runs out; and

l. a set of thirty-three pages of virtual memory: one page is reserved for the kernel domain (for the kernel stack and swapable process context), sixteen pages are reserved for the supervisor (eight pages of I (instruction) space, and eight pages of D (data) space), and sixteen pages are

reserved for the user domain (organized in the same manner as the supervisor domain). Each page is 8k bytes in size and segments may be mapped to each page.

The attributes of kernel supported objects are summarized in figure 3.

Kernel Operations

The following sections will consider the operations provided by the kernel for non-kernel software. All of these operations must satisfy the protection constraints imposed by the kernel supported protection policies. The only exception to this protection is for privileged processes.

Segment Operations

Processes can perform three classes of operations on segments: creation and deletion, modification of attributes, and read or write access in virtual memory.

The operation CREATE_SEGMENT takes the following arguments:

a. the filesystem on which the segment is to be created - the ability to specify the device (filesystem) on which a segment is to be created allows non-kernel software to control the distribution of segments. This control is particularly useful for performance reasons. Unfortunately, it also permits a "storage channel" per filesystem because of the finite quota of storage available per device. In a complete, operational design, provision for the auditing of these resource exhaustion events (we will see another for exhaustion of processes) provides control of the storage channel short of introducing the complexities of secure quota mechanisms;

b. the access level at which the segment is to be created - an unprivileged process can create segments at any access level greater than or equal to its own; however, no information as to the success or failure of the creation request will be returned by the kernel;

c. the domain of the segment, either supervisor or user; and

d. the size of the segment - segment size is taken from the set {512, 1024, 2048, 4096, 8192} bytes.

ATTRIBUTES	OBJECTS		
	Segments	Devices	Processes
Unique Name	48 bits	32 bits	32 bits
Current Access Level	x	x	x
Maximum Access Level	x	x	x
Privilege			x
Current Domain	K,S,U	S	K,S,U
Previous Domain			S,U
User Identity	x	x	x
Group Identity	x	x	x
Discretionary Permissions	x	x	
Value	x	x	
Size	x		
Request Queue		x	
Byte Pointer		x	
Hardware Context		x	x
Priority			x
Trap Vector			x
Timer			x
Virtual Memory			x

Figure 3. Kernel Object Attributes

The kernel will return to the requesting process the unique name (uid) of the segment purportedly created. No status message as to success or failure can be returned. The name must be guaranteed to

be unique and to bear no relation to the sequencing or rate of requests for segment creation. The kernel will derive segment names from the system clock having a resolution on the order of 10 microseconds. The kernel cannot possibly respond to any one request for segment creation in that time, thus guaranteeing the uniqueness of name. The user(owner) of the created segment is the caller, and the group and discretionary permissions are null. Therefore, SET_SEGMENT_DISCRETIONARY_PERM (see below) must always be called after a create operation.

The operation DELETE_SEGMENT allows deletion of any segment (for unprivileged processes the process must have an access level less than or equal to the access level of the segment) for which the requesting process can supply the name (uid). The kernel provides no information to the requesting process as to the success or failure of the operation.

The operation SET_SEGMENT_DISCRETIONARY_PERM allows a process belonging to the user of a segment (or a process privileged to violate discretionary access controls) to alter the discretionary permissions and owner/group identity of a segment. An unprivileged process must have the same access level as the segment. This operation should be used after segment creation to set the (initially null) discretionary permission for the segment.

The operation SEGMENT_STATUS allows retrieval of segment attributes by processes. For unprivileged processes, the process access level must be greater than or equal to the access level of the segment.

The above two operations permit the interrogation and modification of those segment attributes subject to change. The segment attributes of name, size, and access level are invariant.

Processes control access to segments as virtual memory through the operations MAP_SEGMENT and UNMAP_SEGMENT. Each process possesses a set of thirty-two 8k byte pages of virtual memory (accessible to non-kernel software). Each of these pages can be "mapped" with the MAP_SEGMENT operation, which takes four arguments:

a. the name of the segment to be mapped, that is, the segment associated with a portion of the requesting process' virtual memory such that memory accesses to that portion of memory are directed to the segment;

b. the name of the page of virtual memory to which the segment is to be mapped;

c. the access modes the segment is to have (READ-ONLY or READ-WRITE). These modes must be consistent with respect to the privileges of the process and the protection attributes of the process and segment; and

d. the justification of the segment within the 8k byte page, that is, whether the segment is justified LOW (the low order byte of the segment is aligned with the low order virtual address of the page) or HIGH (the high order byte of the segment is aligned with the high order virtual address of the page).

Any number of the processes may map the segment (pending successful access control checking) and they have access to the same copy of the data within the segment. The kernel manages the migration of mapped segments from core to disk and from disk to core.

The association of page and segment is severed via the kernel operation UNMAP_SEGMENT. It takes only one argument, the page name, and all subsequent references to that page will cause memory management traps.

The information associated with each mapped address space register can be retrieved by the REGISTER_STATUS operation. It takes one argument, a register number, and returns the information specified in the last map operation for that register. An error occurs if the specified register is not mapped.

For the operations DELETE_SEGMENT, SET_SEGMENT_DISCRETIONARY_PERM, SEGMENT_STATUS, and MAP_SEGMENT, the kernel must guarantee that the process (except for the timing of the operation) cannot distinguish between the refusal of the kernel to execute the operation due to segment non-existence and the refusal of the kernel to execute the operation due to its inaccessibility to the process. Identical constraints will be placed on process access to devices and other processes.

Device Operations

Three types of operations on devices are supported by the kernel: requests for service, alteration of protection attributes, and status inquiries. Devices, as indicated earlier, are considered to be special cases of processes. In this sense, requests for device service are treated as a special case of InterProcess Communication (IPC) and the same mechanism is used to implement both facilities. To use a device, a process uses a kernel operation which makes a request of the specified device. Devices service their queues of requests logically asynchronously of other devices

and processes. They can be considered to have their own limited instruction set for communication with processes, alteration of protection attributes, and communication with segments. First, the kernel operations on devices will be discussed, then the operations the (asynchronous) devices can perform will be discussed.

Kernel Operations. The operation IO_READ requests a transfer of an array of bytes from the device to an area of the process' virtual memory. The area specified must lie within one segment. The requesting process must specify the number of bytes to be read and the unique name of the device from which they are to be read. In this way, the kernel abstracts all devices to appear as DMA devices.

The kernel also allows the requesting process to transmit a single byte of label to the device that tags the request. The device will echo this label in the IPC message the device will send to the requesting process (assuming the process still exists) signifying that the request has been satisfied or an error has occurred during transfer. This label facility permits the process to have a number of outstanding device requests (256) and thus permits internal multiprogramming within any process.

The requesting process must have the same access level as the access level of the device, discretionary observe and modify access to the device, and have an access level equal to the access level of the specified segment.

Transfers cannot cross page/segment boundaries and are thus limited to 8192 bytes. For terminal class devices, those through which people can interact with the kernel and non-kernel software, read transfers are not only terminated after transfer of a specified number of bytes, but may be prematurely terminated by the entry of an "end_of_line" character.

The IO_WRITE kernel operation requests device action to transfer an array of bytes from a specified segment mapped into the requesting process virtual memory to a specified device. The operation requires the same arguments as the IO_READ operation. A similar IPC message will be transmitted by the device upon completion of the request. Identical access control policies are enforced for the IO_WRITE and IO_READ operations.

The operation IO_FUNCTION requests a non-data transfer operation by the device. These requests are highly device specific. For magnetic tapes, the functions of writing end_of_file marks, rewinding, and moving the tape forward and backward one record are performed by IO_FUNCTION requests. For terminals, the functions of

setting various (UNIX compatible) terminal modes (such as ECHO and RAW) are performed by IO_FUNCTION requests. The same access control constaints that applied to IO_READ and IO_WRITE apply to IO_FUNCTION.

There are two kernel operations available to alter the protection attributes of devices. The first operation is available only to privileged processes, and allows modification of a device access level. The operation is called SET_DEVICE_ACCESS_LEVEL. The access level is modified to the value specified by the argument that must be less than or equal to the maximum access level of the device - which itself can only be adjusted by rebuilding the system. The operation can be performed on only non-terminal devices. The access levels of terminal devices are modified as described below. When a device's access level is changed, the request queue for the device is cleared and the device is placed in a "standard" state, allowing no information to pass via, for instance, the location of the tape currently mounted on the device.

The operation to set the discretionary permissions for a device is SET_DEVICE_DISCRETIONARY_PERM, and is similar to SEG_SEGMENT_DISCRETIONARY_PERM.

Terminal devices have three preemptive "interrupts" that they can cause: the "quit" key, the "interrupt" key, and the "hangup" condition. These UNIX-compatible conditions cause a trap to occur in the process currently using the terminal. Each device can be labelled with the process name (pid) and user identifier of the process "currently using" the device. These values may be set (by any unprivileged process at the same access level) via the operation GET_TERM_INTERRUPTS. When one of these conditions occurs, the last process to do a GET_TERM_INTERRUPTS for the terminal will get the corresponding trap.

Device Operations. As indicated above there exists a second set of device operations: those operations conceptually executed by non-privileged code running "within" the device. These operations include:

a. IO_REQUEST_SERVICE: the "main-loop" of each I/O device, servicing its request list, taking each request in turn, tranferring bytes to/from the devices internal memory to the process virtual memory segment and sending an IPC message to the requesting process (if it still exists) upon completion of the requested service;

b. TRAP: this operation, the "quit", "interrupt", or "hangup" signal, transmits an interprocess preemptive signal to the

process last executing a GET_TERMINAL_INTERRUPTS kernel operation (if the process still exists);

c. DISPLAY_SECURITY_LEVEL: this operation displays the current access level of the device on the device itself, framed by kernel peculiar characters (that is, characters only available to the kernel and not transmittable by non-kernel software - the current prototype reserves the ASCII "bell" character for this purpose);

d. NEW_SECURITY_LEVEL: a terminal initiated operation (again framed by kernel peculiar characters) that alters the current security level of the terminal to the value specified - in addition to altering the current security level of the terminal, the operation also transmits an IPC message to the process having the name of "zero", that is assumed to be the fully privileged "root" process of the system, the message containing the information that the device (identified by its unique name) has changed its security level to the named value; and

e. LOGOUT: a terminal initiated operation (framed by kernel peculiar characters) to set null discretionary access for a terminal, so that no processes can access it until the new_security_level function is executed. This operation is also initiated by turning off or hanging up the terminal.

The operations TRAP, DISPLAY_SECURITY_LEVEL, NEW_SECURITY_LEVEL, and LOGOUT, can only be executed by devices that are interactive terminals, and are initiated by the user at the terminal, rather than by code running in the supervisor domain. The user causes the TRAP function by either hitting the "interrupt" or "quit" key (UNIX-comatible), or by hanging up or turning off the terminal.

The last three requests are made by entering the ASCII "bell" character, which acts as an indicator to the kernel that one of these requests follow.

Process Operations

Process operations supported by the kernel can be divided into the following classes:

a. operations controlling the existence of processes and inquiries thereto;

b. operations facilitating equitable use of processor resources;

c. operations facilitating InterProcess Communications; and

d. operations manipulating the control structure within the virtual environment of each process.

The last class of operation, while having trivial protection constraints, is supported within a security kernel since the virtual facilities provided by the kernel must be securely provided. A primary function of any kernel concerned with controlling access on all "storage channels" is the secure multiplexing of real resources among the virtual resources provided to processes.

The operations concerned with the existence of processes are SPAWN, DIE, and PROCESS_STATUS.

The SPAWN operation creates new processes. Any process can create new processes. Processes have no kernel supported control relationship with their parents, ancestors, or descendants. An unprivileged process can create processes at access levels greater than or equal to its own. An appropriately privilged process can create processes at any level. Process creation is formally considered a modification. The creating process can specify the initial program counter, stack pointer, process privilege (a subset of the creating process' privilege), and a partial address space for the new process. The creating process can specify two segments of the new process' virtual memory; of course the created process must have proper access to them. The unique name of the created process (clock derived) is returned to the creating process, but no indication of the success or failure of the operation can be returned.

DIE is the suicide operation for an existing process. Processes must abort themselves. The process has the responsibility for cleaning up its virtual memory before it expires; the die operation merely unmaps any segments currently mapped, and deletes any unreceived messages.

The PROCESS_STATUS operation is an inquiry request to determine the protection attributes of a process. This operation takes a process name (pid) and returns the access level, and user/group identifiers of the process if it exists and if (for unprivileged requestors) its access level is less than or equal to the access level of the requestor.

The operations providing some untrusted information for the scheduling of processor resources are SET_PROCESS_PRIORITY and DOZE.

The SET_PROCESS_PRIORITY operation allows each process set an integer priority to any integer value. The kernel's only scheduling algorithm is the following: the process with the highest value of priority, that is ready to run, is given the processor. Clearly, the placement of priority control in the supervisor/untrusted code permits exploitation of the timing channel occasioned by competition for the CPU resource; however this kernel does not pretend to address the blockage of these resource competition timing channels.

The DOZE operation allows a process to give away the processor to a higher priority ready-to-run process. The kernel does no preemption of processes based on quantized resource allocation exhaustion. Thus all control of the processor has been distributed to the supervisors of the several processes. In particular, these supervisors, by convention, will allow higher priority processes to run (via the DOZE operation) at intervals governed by whatever scheduling algorithm is deemed equitable. It should be noted (as we will see in the following section) that these supervisors do not communicate with each other, so all decision making is based on the local performance of each process and global parameters embedded in the programs of each supervisor. No insecurities result.

The DOZE operation has an additional parameter: an eligibility flag. The set of processes is partitioned into two sets at any one time: the "eligible" set - those processes able to actively compete for CPU and memory resources, and the "ineligible" set - those processes temporarily barred from competition either based on a low value of priority or because they are waiting for an unsent IPC message. The kernel will only support a small number of eligible processes (about 5 for 256k bytes of primary memory) based on the amount of resources available. This figure will be a compilation constant. When a DOZE is specified, the requesting process can remain eligible, thus giving up the CPU to some other eligible process, or it can become ineligible, allowing another ineligible process to become eligible.

The DOZE operation has one final argument: an amount of time. If a process chooses to become ineligible, he may further request that he remain ineligible for a specified amount of (real) time. The kernel guarantees that he remain ineligible for at least that amount of time.

InterProcess Communication is accomplished via two mechanisms, serving differing purposes. The first mechanism is non-preemptive and transmits 16 byte messages between processes. The kernel provides a routing and buffering mechanism for these messages. The second mechanism is a pseudo-preemptive one, specifically designed to allow communication of terminal interrupt-like signals to

processes using those terminals. The first mechanism is supported by three operations: SEND, RECEIVE, and INQUIRE.

The SEND operation sends a 12 byte arbitrary message to a named process. SEND is formally considered a modification and thus obeys the mandatory access controls imposed on all modifications. The kernel will reliably append the name of the sending process onto the message before receipt. Messages are internally queued within the kernel pending receipt. Exhaustion of kernel message buffer space will cause a system crash, but should not occur with appropriately programmed supervisor mode programs.

The RECEIVE operation operates on the message queue associated with each process. If messages are queued it will return the first queued message and delete it from the queue. If no messages are currently queued, processor control will be transferred to another process pending receipt of a message. The process can optionally specify its desire to remain eligible during a wait for a message as in a DOZE operation. The process can also specify an increment of real time which can expire before an error return caused by a wait for message and no message arrived during the specified interval of real time.

The INQUIRE operation is a non-waiting variant of RECEIVE that will give an immediate error return if no messages are queued for the process.

The above mechanism permits malicious software to crash the system (at will) through indiscriminate numbers of InterProcess Messages. The responsibility for such denial of service concerns has been placed in the supervisor domain of each process.

The preemptive form of IPC takes the form of the operation IP_TRAP. IP_TRAP (having the same access constraints as SEND) can signal to another process the existence of a preemptive condition. The kernel supports up to sixteen such conditions, some of which are preallocated to the hardware conditions of trap-like situations. The IP_TRAP operation notifies the destination process of the occurrence of one of these conditions and the existence of the condition is recognized upon the next return from a kernel operation. The format of the recognition will be discussed in the next section on intraprocess trap handling.

The final category of process control operations concerns the manipulation of a process' own virtual environment. Two facilities are provided: timers and clocks, and preemptive traps.

Four time handling mechanisms are provided. The first is the operation RTIME that simply returns the value of the system-wide calender clock. The second is the operation VTIME that returns the value of the process-local version of the system clock - a virtual real-time clock that records the elapsed time the process has been in execution outside of the kernel. The non-kernel software has no explicit control over either of these mechanisms. The third, and last timer, is a virtual interval timer. The non-kernel software, via the operation ENABLE_TIME_TRAP, can set a value of virtual time (in units of about 1 millisecond) after which a preemptive timer run down trap will be caused.

The preemptive trap mechanism is an image of the hardware provided trap mechanism. The occurrence of a trap, either caused by IP_TRAP or by intraprocess actions, will result in two actions:

a. the placement on the supervisor stack (position of the supervisor stack pointer register) of the current ps, pc, an integer labelling the type of trap, and several words of descriptive information that is process local and trap type dependent (viz., for a memory management trap the contents of the CPU MMU control registers); and

b. the kernel will return into supervisor space through location 0 in supervisor I-space that is assumed to contain the first instruction of a trap handling routine.

Note that if either no segment is mapped to process page 0 (containing the supervisor trap vector) or insufficient space, existence, or access is available on the supervisor stack, the kernel will simulate a die operation for the process.

The last time handling function, STIME, allows the caller to set the value of the system clock. To avoid a covert information channel through the clock, the STIME operation changes the system clock only the first time it is called.

Figure 4 contains a summary of all the kernel operations discussed in this section.

Kernel/User Communications

All communication with the kernel must be mediated by the supervisor domain of the non-kernel process software. Kernel operations are invoked via the hardware TRAP instruction executed within the supervisor domain. Arguments to kernel operations are pointed to via the supervisor stack pointer within supervisor virtual space. It is the responsibility of the supervisor software

SEGMENT	Create and Delete	CREATE_SEGMENT DELETE_SEGMENT
	Modify Attributes	SET_SEGMENT_DISCRETIONARY_PERM
	Status Inquiry	SEGMENT_STATUS
	Virtual Memory Access	MAP_SEGMENT UNMAP_SEGMENT REGISTER_STATUS
DEVICE	Service Requests	IO_READ IO_WRITE IO_FUNCTION
	Modify Attributes	SET_DEVICE_ACCESS_LEVEL SET_DEVICE_DISCRETIONARY_PERM
	Status Inquiry	DEVICE_STATUS
	Accept Terminal Interrupts	GET_TERM_INTERRUPTS
PROCESS	Create and Delete	SPAWN DIE
	Status Inquiry	PROCESS_STATUS
	InterProcess Communication	SEND RECEIVE INQUIRE IP_TRAP
	Processor Resource Control	SET_PROCESS_PRIORITY DOZE
	Virtual Environment Manipulation	RTIME VTIME ENABLE_TIME_TRAP STIME

Figure 4. Kernel Operations

to guarantee that sufficient stack space is available for the kernel to fetch arguments and return results. The kernel always carefully validates that necessary supervisor stack space is available and will abort the process if sufficient space is not available.

KERNEL INTERNAL ARCHITECTURE

Complete documentation of the implementation of the kernel is not the intention of this discussion. This section will only attempt to sketch the kernel architecture. We will approach the architecture from two directions. The first direction considers the decomposition of the kernel into levels of abstract machines, the totality of which define the operations of the "top level" of the kernel. The second direction can be considered an orthogonal decomposition. The kernel, as indicated previously, is internally multiprogrammed. Certain levels of abstract machine are responsible for the definition of the mechanism for multiprogramming support: the levels of process definition. The second decompostion will indicate how these processes are used in the construction of other abstract machines within the kernel. The first decomposition will give the reader some intuition into how the facilities of the kernel are functionally constructed. The second decompostion will give the reader some intuition into the architecture of their construction.

Abstract Machine Decomposition

The first uses of hierarchical decomposition into abstract machines assumed a linear ordering of abstraction. Unfortunately, experience does not support this notion. The set of abstract machines forms, at best, a partial ordering. Most machines do not use all predecessor machines in the same manner that not all programs use all features of the language in which they are written. The following discussion will attempt to define this partial ordering for the kernel, which is diagrammed in figure 5.

Interrupt Control

The lowest level of the kernel controls the hardware interrupt mechanism. All software executes at one of two interrupt levels: level 0 or level 7. The normal mode of execution is level 0. Only a small portion of the kernel ever executes in level 7, blocking all external interrupts. In particular, the queue managers (message and character queues) and the primitive process abstract machine make use of this module.

Two operations are provided: the first transitions the processor to level 7 and returns the previous hardware processor

ABSTRACTION LEVEL	MAJOR MODULES
USER TRAP LEVEL	User Trap Manager Top-Level Kernel Function Modules
USER PROCESS LEVEL	User Process Manager User Device Managers
ELIGIBLE PROCESS LEVEL	Eligible Process Module Segment Manager Memory Access Module Memory Management Module
PRIMITIVE DEVICE LEVEL	Primitive Device Managers
PRIMITIVE PROCESS LEVEL	Primitive Processss Module Primitive Event Module Primitive Message Port Module
SUB-PROCESS LEVEL	Clock and Timer Manager Character Queue Module Message Queue Module System Abortion Module Bit Pool Management Module
COMMON FACILITIES/ EXTENDED TYPES LEVEL	Multiple Precision Integers Bit Vectors Access Levels Hash Tables Stacks Queues Doubly-Linked Queues Priority Queues
INTERRUPT CONTROL LEVEL	Lock Out Interrupts Allow Interrupts

Figure 5. Kernel Abstract Machines

status as value, and the second restores the processor status to the value passed as argument. Thus, for this module, the invoker of the operations has the responsibility for storing previous values of processor status/priority.

Common Facilities/Extended Types

The next level of the kernel, though not dependent on the interrupt control module, defines an assortment of extended data types for use throughout the rest of the kernel. Each of the extended types is explained below.

Multiple Precision Integers. Operations are defined for double (32 bit) and triple (48 bit) precision integers. These operations include zeroing, moving, adding, substracting, multiplying, dividing, incrementing, decrementing, shifting, and comparing.

Bit Vectors. Arbitrary length bit vectors can be manipulated, including bit addressing, setting, clearing, testing, and finding the first "1" bit in a vector.

Access Levels. The access level data type is defined, and operations for initializing, moving, and comparing access levels are provided.

Hash Tables. The hash table data type defines an associative memory organization on threaded lists, allowing the association of each element with a triple precision integer "key" value. The Operations defined on hash tables include initialization, adding elements, removing elements, and looking up elements.

Stacks. Last-in-first-out stacks are defined, along with operations to push and pop integer values.

Queues. First-in-first-out queues are defined, along with operations to put integers into a queue, and remove integers from the end or middle of the queue. Because the queue is only singly threaded, removal from the middle is not very efficient.

Doubly-Linked Queues. These queues are similar to normal queues except that they are doubly threaded, so removal of an element from the middle of the queue is more efficient.

Priority Queues. These queues are lists of integers maintained in order of increasing value of an integer priority associated with each list element. When elements are added to priority queues, their position depends on their priority. Other operations on priority queues include those to remove the highest priority element in the queue, and to remove any arbitrary element of the queue.

Sub-Process Level

The next kernel level is a collection of modules that make use of the extended types and interrupt control modules to create an environment rich enough to support the lowest level of processes in the kernel. The functions of these modules are explained below. Note that these modules do not depend on each other, and are in no way related.

Bit Pool Management. This module makes use of the bit vector management module to provide a mechanism for the definition and use of bit pools: a management mechanism for 1M bytes of storage allocated using the "buddy" system of management [8] in units of 512, 1024, 2048, 4096, and 8192 bytes. Operations are provided for the allocation and return of blocks of storage of each of these sizes.

System Abortion. This module makes use of the interrupt control module. It provides a mechanism for halting the system and displaying 16 bits of data on the console lights as the cause of the system crash.

Message Queues. This module makes use of the stack, queue, and interrupt control modules to construct a module for the definition and use of message queues. Message queues are a mechanism for the queueing of 16 byte messages of arbitrary value. Messages are allocated from a common pool of storage (fixed at kernel compilation time) and threaded through the list rooted in the message queue definition itself. Exhaustion of messages causes the system to abort. Operations provided include:

a. initialization of a new message queue;

b. placing a new message on a named message queue; and

c. removing the first message from a named queue.

All operations are performed at interrupt level 7.

Character Queues. This module makes use of the same modules as the message queue module but the value queued is only one byte in size. Further, exhaustion of the kernel provided pool of character storage does not crash the system. Rather, attempted queueing of a character when the queue is full causes the character to be lost.

Clock and Timer Manager. The clock and timer manager provides the abstraction of a system-wide calender clock and interval timer from the programmable clock supplied with the system. The clock

provided by this module has a size of 48 bits with a resolution of 10 microseconds. The interval timer provided by this module has a size of 15 bits and a resolution of 1.28 milliseconds. The clock manager executes at interrupt priority level 7.

The system clock is automatically incremented so the only operations on the clock are to read it and to set it. Since there is no security level associated with the clock, and since it can be read by any process, allowing it to be set by any process would provide a storage channel with a high bandwidth. Furthermore, it is desirable to let the clock be set by non-verified code, because the algorithm for computing the clock value is a complex one and would be difficult to verify. Therefore, the clock provided by this module can be set by any process, but it can be set only once. It will normally be set as part of the system startup procedure.

The interval timer can be read and set by any process. The interval timer is decremented at the time of a clock tick if the processor was not running in kernel mode at the time of the tick. Thus the presence of the kernel execution time is hidden. If, when decremented, the timer becomes zero, the timer manager simulates a "timer trap" and the User Trap Manager is invoked as with every other trap.

Primitive Process Level

Primitive processes are the first level of process abstraction within the kernel. This level contains the modules which define them and the operations that can be performed on them. Primitive processes are provided for three purposes:

a. to encapsulate the software defining/managing user processes (to be defined later);

b. to encapsulate the handlers for devices so as to provide efficient servicing of terminal (character-at-a-time) requests for service; and

c. to encapsulate certain kernel facilities so as to simplify the construction of concurrency within the kernel.

Three major data structures and modules are defined at this level:

a. primitive processes: primitive processes each possess a stack (of fixed size), an integer valued priority for obtaining the processor, a name, a program counter, a set of general registers (including stack pointer);

b. primitive events: implementations of semaphores. They are used to control the interaction of primitive processes. Events are implemented using integers and queues; and

c. primitive message ports: a mechanism for the coordinated exchange of messages (defined by the message queue module) between primitive processes. Message ports consist of a primitive event and a message queue. Primitive processes may declare an instance of a message port, initialize it, send messages over it, and receive messages over it.

It is the intention of this level to entirely eliminate external interrupts (other than timer interrupts handled by the clock/timer manager) and convert all external interrupts into signals (V operations) on events assigned to each hardware element that can cause external interrupts. Primitive processes are provided operations to wait for the future or past occurrence of an event that is signalled by some other primitive process. Operations on either primitive processes or events are always performed at interrupt level 7.

The scheduling of primitive processes is implemented using a priority queue of primitive processes that are ready to run (that is, defined and not waiting on a event). Scheduling is preemptive, that is, if during the execution of a low priority process an event is signalled that makes a higher priority process ready to run, the higher priority process will gain the processor until such time as it again waits for an event. An operation is defined for the initialization of primitive processes. They are not dynamically created and defined, rather, they are initialized when the kernel is initialized and remain in existence while the kernel is in existence.

The Primitive Process Table is the kernel data base that defines primitive processes.

The primitive process level also includes the interrupt interceptor module, which is an assembly language module that converts external interrupts to "V" operations on events (signals).

Primitive Device Level

The next level of the kernel is the primitive device level, which encompasses all of the primitive device managers of the kernel. Primitive devices are associated one-to-one with UNIBUS controllers. A multiplexed controller is defined as one primitive device with several other ("user") devices attached to it. The Primitive Device Table names each device controller configured into

the system, and associates with each such controller the following data objects:

a. an event to be used as a lock on the controller mediating access to the controller by several primitive processes;

b. an event to be used by the interrupt interceptor module, to be signalled on input interrupts; and

c. and event to be used by the interrupt interceptor to be signalled on output interrupts.

The structure of the different primitive device manager processes will be discussed in the process decomposition section.

Eligible Process Level

The next kernel level is the eligible process level, which is also the next level of processes within the kernel. Eligible processes introduce the concept of a per-process virtual environment and as such include many modules in the level to make up that environment. The various modules in the level are somewhat interdependent, and also depend on lower levels of the kernel. The modules that make up this level will be discussed in a somewhat hierarchical manner, but the hierarchy is not a true one in that there are some interdependencies.

Memory Management Module. The Memory Management Unit (MMU) hardware provides 48 virtual memory pages divided into 8 instruction pages and 8 data pages for each processor mode: kernel, supervisor, and user. All of the kernel instruction page registers and six of the kernel data registers are reserved to address the kernel code and data that is always resident in memory. The remaining page address registers may be dynamically loaded by the kernel to augment its memory environment and to provide a virtual memory environment for the supervisor and user mode of each process.

The memory management module defines the operations on the MMU hardware and the layout of the virtual memory accessible to processes. This module provides an abstraction of the remaining 34 page address registers: two pages of kernel data space, eight pages of supervisor instruction space, eight pages of supervisor data space, eight pages of user instruction space, and eight pages of user data space. Each page of virtual memory has associated with it a descriptor register. This module provides the mechanism for loading and clearing these 34 MMU registers. Each MMU register has the following attributes:

a. an address into physical memory of the start of the page - expressed in units of 512 bytes;

b. a length of the page - expressed as n where the size of the page is two to the n+9 bytes;

c. a justification for the physical page within the virtual address space of the page (either high or low);

d. access mode - expressed as NULL, READ-ONLY, or READ-WRITE; and

e. access control - hardware maintained flags indicating whether the page has been referenced or written.

This module associates with each of the 34 MMU registers the Active Segment Table (AST) entry associated with the segment addressed by the register. It is thus tightly coupled to the Segment Manager Module, to be defined later.

The 32 supervisor and user registers are used to create the virtual memory environment of those domains. The two remaining kernel registers are used to address the stack the kernel runs on when servicing a user kernel request, and for use by the Memory Access Module.

Memory Access Module. This module uses one of the two free kernel data registers to access data anywhere in memory. It provides two forms of memory access:

a. word-at-a-time access, to allow reading and writing of any arbitrary word in physical memory, for use by device drivers; and

b. access to any segment by the kernel, used at various places to be defined later.

Segment Manager. The Segment Manager provides the definition and mechanism for the concept of a segment: a virtual memory information repository. The Segment Manager provides the following operations:

a. the creation of segments on filesystems;

b. the "activation" of segments - allowing a fast access copy of the segment defining information to be kept in core for use by an eligible process;

c. the "binding" of segments - bringing the segment into core for access through the page descriptor of some eligible process (and eventually through the hardware page descriptors);

d. the "deactivation" of segments - permitting segments to migrate their defining information back to the filesystem repository.

e. the deletion of segments.

The primary tasks of the Segment Manager are the control of primary memory: the determination and transport of segments to and from secondary storage and primary storage; and the control of secondary memory: the creation and deletion of segments on filesystem and the necessary space management questions involved.

The primary data bases used by the segment manager are the Active Segment Table (AST) and the Core Allocation Table. The Active Segment Table is a hash-associative memory containing in-core copies of the descriptors of activated segments and deactivated segments whose descriptor space in the AST is not required by another segment. The segment manager will preempt the AST slot of an inactive segment in order to "bring in" a copy of the defining information for a segment becoming active.

The AST must have sufficient entries to at least accommodate all concurrently active segments. The size of the AST is determined by multiplying the number of pages per eligible process (32) times the number of eligible processes. This number results from the architectural constraint that all segments mapped to pages of eligible processes must be able to be activated.

A primary subsystem of the Segment Manager, the Core Allocation Module (CAM), operates on the Core Allocation Table. Core is organized as a buddy system of allocation in the sizes specified for pages and segments: 512, 1024, 2048, 4096, 8192 bytes. The CAM allows the preemption of the space allocated to segments, the copying of modified segments back to secondary storage, and the reallocation of the preempted segment to a new segment being brought into core. The CAM also permits the "wiring" of segments such that their primary memory space cannot be preempted. The CAM also permits the "unwiring" of previously wired segments. The CAM implements a "preemptable buddy system allocation mechanism".

When not in primary memory, segments reside in secondary storage. Secondary storage is organized by the kernel into "filesystems". The management of each filesystem is assigned to a

"filesystem manager", which is a primitive device manager. Each filesystem manager is identified by the Filesystem Table, indexed by filesystem name (1 byte). Each filesystem manager defines a Table Of Contents (TOC) on its filesystem containing the following data elements:

a. a buddy pool-organized free space data base (implemented by the bit pool module); and

b. an array of segment descriptors containing the defining information (name, access level, user/group, permissions, and secondary storage address) for each segment defined on the filesystem.

Each filesystem manager supports the following operations:

a. creation/deletion of segments;

b. read/write of segment descriptors; and

c. read/write of segment bodies to/from primary memory.

The Segment Manager itself has a distributed portion and a centralized portion. The distributed portion runs in each eligible process and makes requests of the centralized portion, which resides in a primitive process. The requests are sent to the segment manager primitive process via a primitive message port. When the segment manager has finished processing the request (which it handles on a first-come-first-served basis), it sends a response message over an eligible message port (to be defined) associated with each eligible process. Thus, after the distributed portion of the segment manager sends a (primitive message) request to the centralized portion, it waits for a reply on an eligible message port. Waiting on an eligible port allows other eligible processes to run if they are ready.

If the segment manager primitive process needs service from a filesystem manager, it sends a primitive message to the filesystem manager and waits for a primitive message reply. Similarly, filesystem managers request service from other filesystem managers.

Eligible Process Module. The eligible process module defines the data structures and operations necessary for creation of eligible processes. Eligible processes introduce the concept of a per-process virtual memory. They also encompass a resource "governor" in that the number of eligible processes defined is coupled to the amount of virtual memory resources (essentially

primary memory) available. Each eligible process possesses the following attributes:

a. a set of virtual address registers, each containing a description of a page of virtual memory - each eligible process has control of 1 kernel page to be used for a kernel domain stack for each eligible process, 16 supervisor pages, and 16 user domain pages (each eligible process can execute in all three domains - all eligible processes share the other seven kernel pages containing kernel code, data, and device registers);

b. context similar to primitive processes - stack pointers (one for each domain), a program counter, and a set of general registers;

c. a name; and

d. a priority.

The data base that contains this information is the Eligible Process Table.

All eligible processes are multiplexed onto one primitive process dedicated to this purpose. This primitive process is the lowest priority primitive process, so that eligible processes run only when no other primitive processes need the CPU. Eligible process scheduling within this primitive process is done in a manner similar to primitive process scheduling, though non-preemptive.

The eligible process module contains the definition of, and operations on, eligible events, to be used for the coordination of eligible processes. Since eligible process scheduling is non-preemptive, an eligible process will execute until it waits on an eligible event. An eligible event itself is made from a primitive event (which acts as a lock on the eligible event), and integer, and a queue.

The eligible process module also contains the definition of, and operations on, eligible message ports. An eligible message port, similar to a primitive message port, is used to send messages to eligible processes. An eligible message port is composed of a primitive event lock, a message queue, an eligible event to represent the existence of messages, and a queue of eligible processes awaiting messages.

The eligible process module manages the mapping of the eligible process virtual address space to the machine provided by the Memory

Management Module. Operations are provided for the loading/clearing of the eligible processes virtual page descriptors and for the translation of these operations into operations on MMU registers (as perceived by the MMU Management Module).

User Process Level

The next level of the kernel is the user process level, which is also the top level of the process hierarchy within the kernel. The modules at this level are the User Process Manager and the User Device Managers.

User Process Manager. The eligible process level defines a set of eligible processes which are multiplexed within a primitive process. Similarly, the user process level defines a set of user processes that are multiplexed among the eligible processes. A user process that is not currently assigned to an eligible process is said to be ineligible.

The process of transitioning user processes between the eligible and ineligible states is performed by a process called the User Process Manager, or UPM. The UPM fills eligible processes with user processes when the eligible processes become free.

When an eligible process wishes to become ineligible (because the supervisor issued a DOZE or RECEIVE kernel call and so indicated), the eligible process sends an eligible message to the UPM (which is the highest priority eligible process). The UPM then copies the eligible process' context into its user process context, and places the user process on a queue of dozing or receiving processes. Similarly, if an eligible process wants to die (because the supervisor made a DIE kernel call), he sends a message to the UPM. The UPM then clears out the eligible process and deletes the corresponding user process.

In either of the above two cases, the UPM has freed an eligible process. The next step is for the UPM to select a new user process to make eligible. This selection is made in the following steps.

a. A queue of dozing user processes is checked to see if any have dozed for the specified amount of time. If any have, the process is removed from the dozing queue and placed on the ready (priority) queue.

b. A queue of receiving processes is checked to see if any have either received an IPC message or timed out. If any have, the process is removed from the receiving queue and placed on the ready queue.

c. The highest priority process on the ready queue is selected to be made eligible.

User processes have the following context:

a. a kernel stack segment - the kernel stack segment is always core resident when the process is eligible for execution;

b. a message queue of IPC messages;

c. a set of timers, etc.; and

d. a virtual memory consisting of segments mapped to supervisor and user domain pages.

When a process becomes eligible, its context must be copied to the context of an available eligible process and the process can then proceed to execute. A user process is made eligible by the following operations:

a. its kernel stack is activated and wired into core; and

b. its context (stack pointers, general registers, interval timer) is copied into the eligible process context.

The kernel domain of each eligible process will respond to segment faults of mapped segments by attempting to "activate" the segment and "bind" it into core and load the page descriptor register for that segment. There is no guarantee as to the amount of time the segment will remain in core.

A user process is made ineligible by the following operations:

a. its mapped segments are deactivated;

b. its context is copied from the eligible process back to the User Process Table entry for the process; and

c. its kernel stack is unwired and deactivated.

The primary data base administered by the User Process Manager is the User Process Table containing one entry for each defined user process. The table is organized as a hash table.

User Device Managers. The other modules at the user process level are those that implement user device managers. A User Device is any supported device not containing a filesystem. The "user" interface to devices was defined in a previous section. User

Devices appear to be special purpose processes. User Devices are defined in the User Device Table, initialized at kernel initialization time. This table contains the following information:

a. the definition of the mechanism necessary to communicate with the primitive processes managing each user device;

b. the data defined for devices at the kernel interface (viz., access level); and

c. the Primitive Device Table entry for this device: more than one user device may share a common Primitive Device - use of the controller is coordinated via the primitive event in the PDT.

Each User Device Manager runs as a primitive process, and receives its requests for service via a primitive message port. The requests for service are processed and completion IPC messages are sent to the requesting process. The functionality of terminal and non-terminal user device managers will be discussed in the section on process decomposition.

User Trap Manager

The highest level of the kernel is the User Trap Manager. This module handles intraprocess trap conditions vectored to the kernel by the hardware trap vector initialized at kernel compilation time. An interval timer trap can also be simulated and vectored to the user trap manager by the clock and timer manager.

The user trap manager is invoked when some form of hardware trap occurs or when the interval timer trap is simulated. Three generic types of traps are handled by the user trap manager, and discussed below.

Kernel Invocation. A hardware TRAP instruction executed in the supervisor domain is interpreted by the user trap manager as a kernel invocation. These traps are vectored to a reentrant subroutine that performs access checking, and if successful, performs the operation requested by the software executing the supervisor domain. All access checking occurs within these trap manager subroutines. After the kernel operation has been performed, control returns to the supervisor domain after the TRAP instruction, unless there are any other trap conditions that should be reflected to the supervisor, as explained below.

Reflected Traps. Certain trap conditions are not processed by the kernel but rather are reflected to the supervisor domain by

placing information about the trap on the supervisor stack, and forcing execution to continue at location zero in the supervisor domain. Traps that are reflected in this manner include:

a. any trap specified by an IP_TRAP kernel call;

b. interval timer rundown traps;

c. hardware traps conditions such as bus errors, invalid instructions, floating point exceptions; and

d. certain types of MMU faults, particularly access violations, and registers that have not been mapped by the supervisor.

Page Faults. There is one type of MMU fault that is not reflected to the supervisor. This occurs when the supervisor has mapped a page, but the page is not yet bound into memory. In this case the kernel binds the segment into memory, loads the appropriate MMU register, and restarts execution after the fault. This page faulting mechanism is invisible to the user.

Process Decomposition

The last section described the architecture of the kernel in terms of a decomposition of the kernel into a set of partially ordered levels of abstraction. This decomposition demonstrated the levels of processes within the kernel. This section will now outline in more detail the architecture of the special processes, primitive and eligible, running within the kernel. This section considers the three primary uses of processes within the kernel:

a. the management of user accessible devices;

b. the management of virtual memory; and

c. the management of the transport of user processes to and from an "eligible" state.

User Device Management Architecture

The architecture decentralizes the management of user devices into primitive processes. One primitive process is defined for each user device as its User Device Manager. Requests for service by the device are passed to its manager as a primitive message through a primitive message port dedicated to the user device in the UDT. The requests to a user device manager are sent by the user trap manager in response to a kernel call. If any segments are to be involved in

the I/O request, the trap manager will wire them into memory before the message is sent to the user device manager.

The device manager process utilitizes the PDT to access the hardware registers for the device, and possibly accesses the segment that was wired by the trap manager.

For DMA devices, the process merely translates the virtual address of the transfer into a physical address and loads the device registers; for non-terminal character oriented devices (such as a line printer), it accesses memory using the Memory Access Module) and passes each byte to the device. Figure 6 diagrams the user device manager process structure for non-terminal devices.

For terminals, a slightly different architecture is employed. Terminals require three primitive processes to manage them since:

a. they are allowed to initiate (or their user is) a change in access level;

b. they can initiate input (type ahead); and

c. they require character echoing.

Thus the following three processes are defined for terminals.

a. The user device manager process responds to requests for service by eligible processes, accesses the device through the processes defined below, and accesses virtual memory.

b. The Terminal Input Manager (TIM) is a device manager at the primitive device management level. TIM waits for input from the terminal (waits on the primitive input event in the PDT), processes change of security level requests by the user, and queues all other characters on two character queues: the first goes to the device manager to satisfy process requests; the second is directed to TOM for echoing.

c. The Terminal Output Manager (TOM) is also a device manager at the primitive device level. TOM waits on a primitive character port for either echoed characters from TIM or output characters from the device manager in response to a process request.

Figure 7 diagrams the process structure terminal device managers.

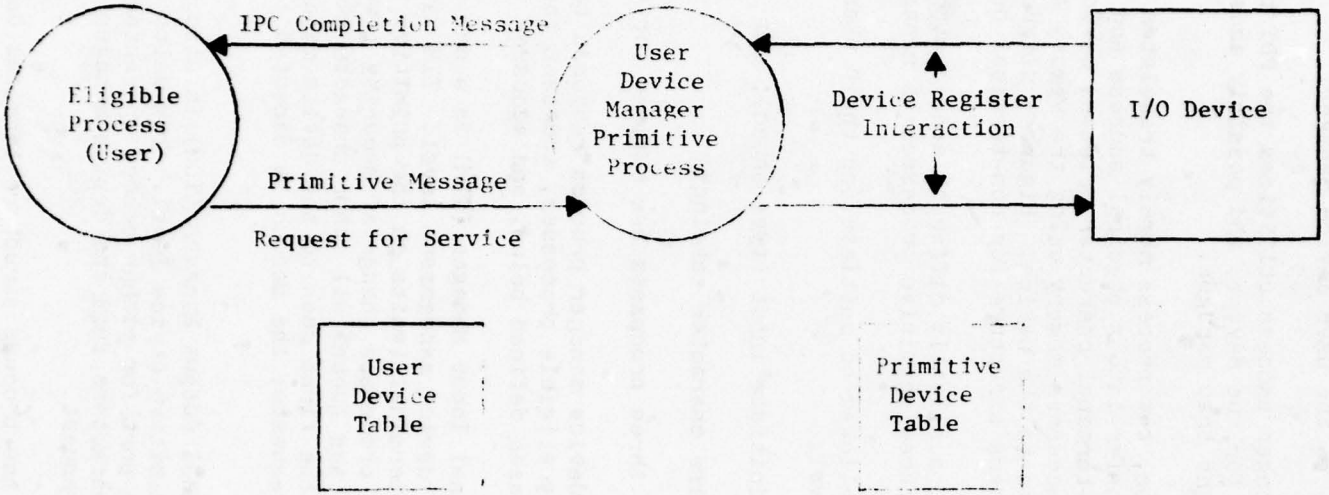


Figure 6. Non-Terminal Device Manager Process Structure

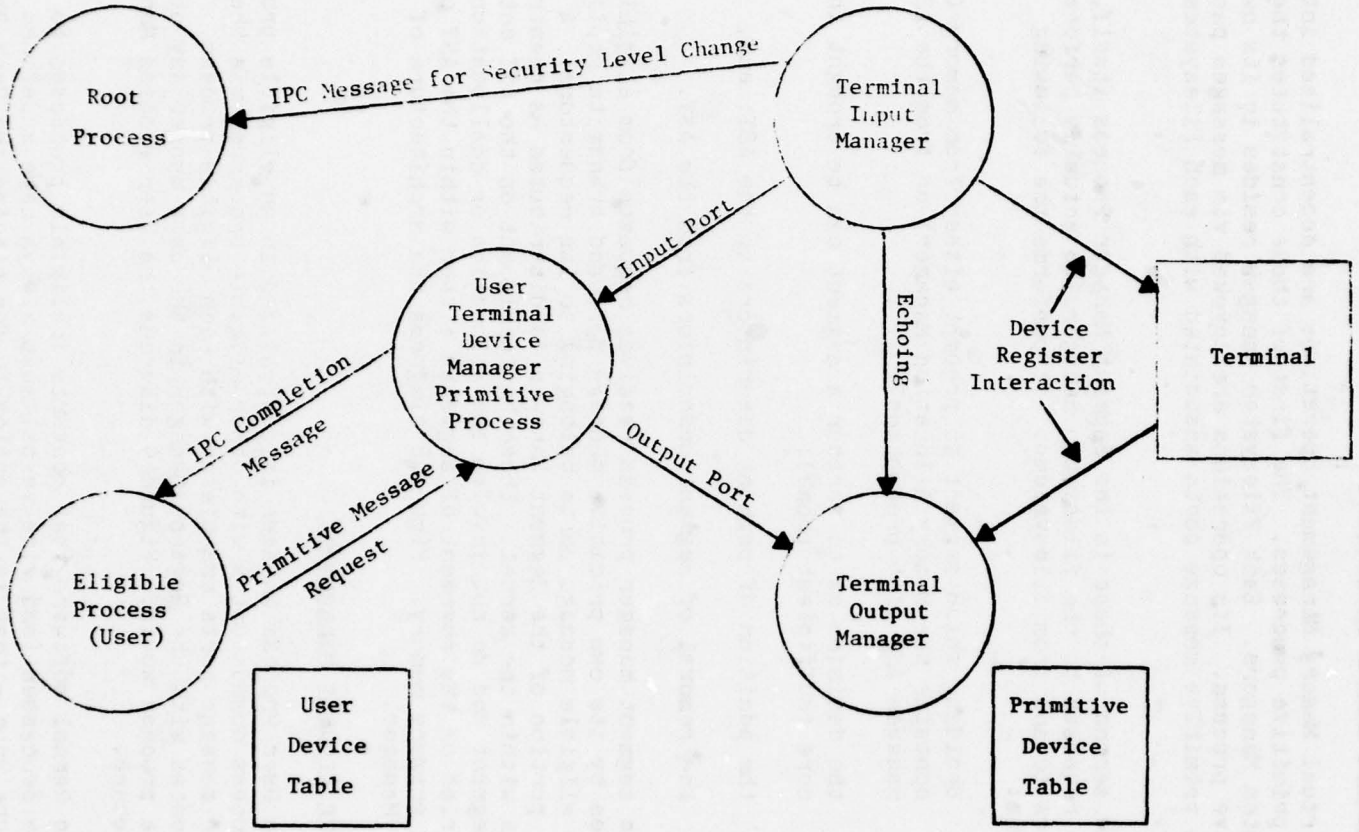


Figure 7. Terminal Device Manager Process Structure

Virtual Memory Management

Virtual Memory Management operations are decentralized into a set of primitive processes. The first of these constitutes the Filesystem Managers. Each Filesystem Manager resides in its own primitive process. Its operations are invoked via messages passed through primitive message ports associated with each Filesystem Manager.

The second of these is the Segment Manager Process itself. It directs requests to the filesystem managers to actually perform the transfers to and from filesystems. It performs the following functions:

a. deciding which segment to preempt either from memory (it contains the Memory Allocation Manager) or from the AST (it manages AST slot preemption);

b. the decision as to whether a segment can be brought into core (sufficient room?);

c. the addition of segment descriptors to the AST; and,

d. the removal of segment descriptors from the AST.

The segment manager process receives requests from eligible processes by its own primitive message port and transmits replies via the eligible message ports belonging to its requestors. A certain portion of the Segment Manager is distributed as reentrant routines within the kernel. These basically act on the AST entry for a segment and do not involve the allocation or deallocation of AST entries or the movement of segments either within the AST or to or from primary memory. Figure 8 diagrams the architecture of the Segment Manager.

User Process Management

The User Process Manager is centralized in an eligible process. This process communicates with other eligible processes via the eligible message ports associated with each eligible process. It communicates with the Segment Manager in the same manner any other eligible process would. Figure 9 diagrams the User Process Manager Architecture.

The kernel software that converts ineligible processes to eligible processes (and vice versa) must always have a defined stack to execute upon - that is the rationale for placing the User Process Manager in a process. The rationale for placing it in an eligible

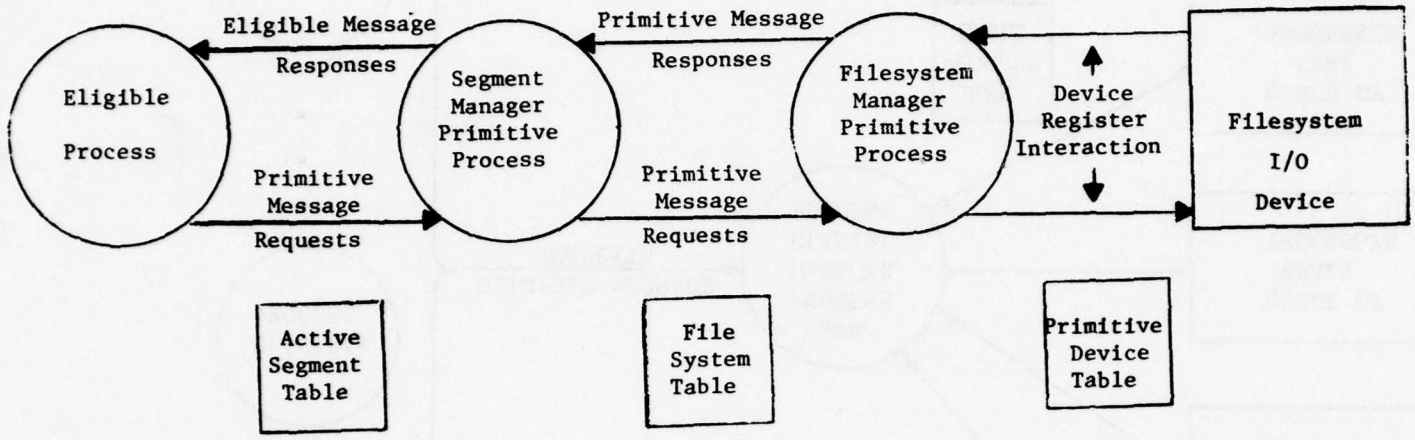


Figure 8. Segment Manager Architecture

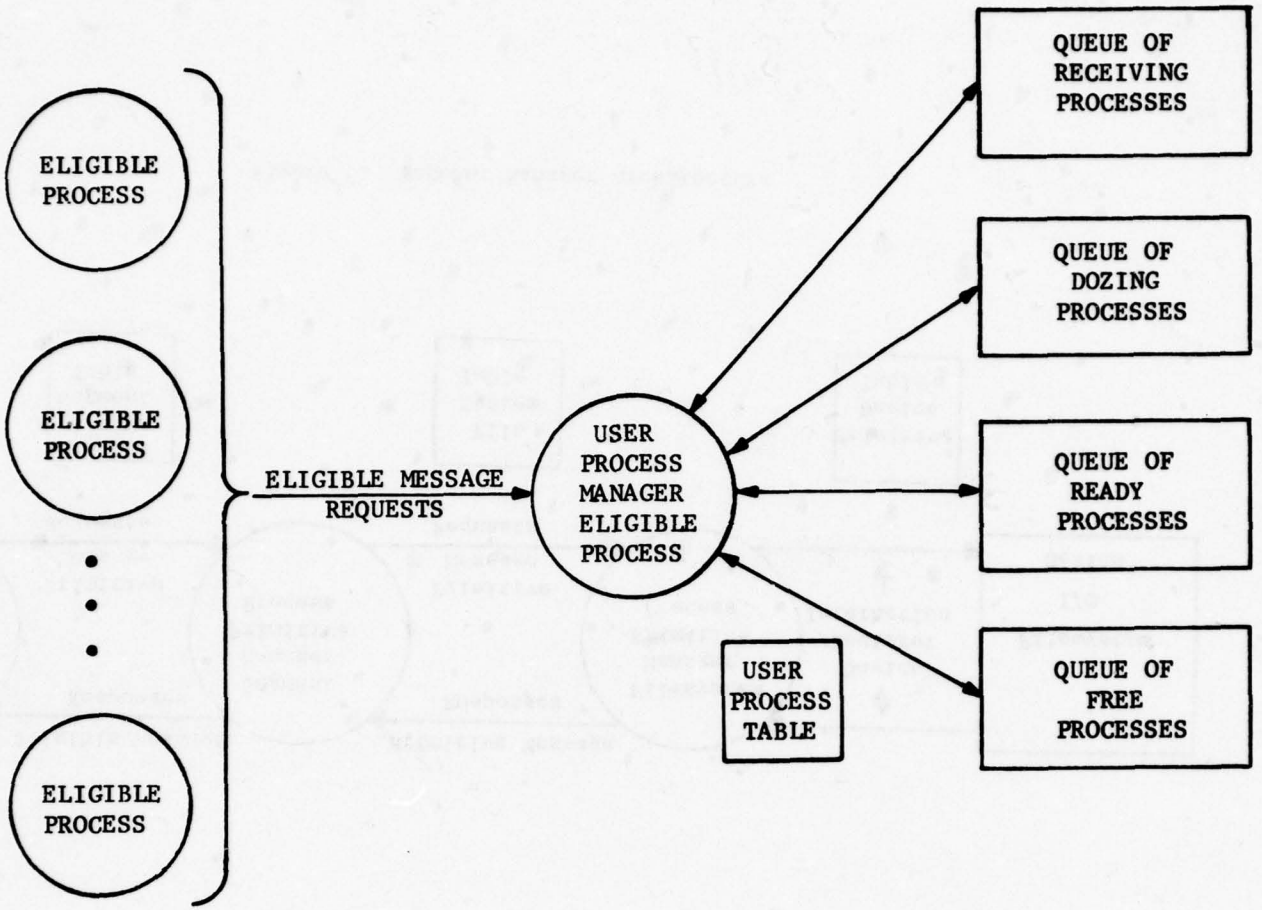


Figure 9. User Process Manager Process Structure

process is to ease the task of communication with other eligible processes and to allow it to communicate with the segment manager.

SECTION IV

PRIVILEGED SUBSYSTEMS

Privileged subsystems are user processes that execute certified, protected code to provide secure, multilevel facilities that cannot be conveniently furnished by the kernel. In particular, they are appropriate for performing such operations as user process creation, login, and multilevel message queuing. These tasks are handled by the root process, the discretionary authenticator, and port manager, respectively. This section will discuss the protection issues of privileged subsystems, and will discuss each of the subsystems in more detail.

PROTECTION ISSUES

Privileged subsystem processes make up the Trusted Subjects [7] mentioned earlier, subjects privileged to violate one or more of the following access control policies:

a. the simple security condition;

b. the security *-property;

c. the simple integrity condition;

d. the integrity *-property; and

e. the discretionary protection mechanism.

The need that the root process, discretionary authenticator, and port manager have for such privileges will be discussed in the following sections.

Since improper operation of software in privileged processes can cause a security compromise, the software must be verified, in a manner similar to security kernel software (though easier to accomplish), and protected from intentional or accidental compromise. This protection is provided by the integrity policy supported by the kernel. Privileged processes's code segments have very high integrity levels.

ROOT PROCESS

At system initialization, after the kernel is initialized, the root process is started by the kernel. The root process executes wholly in the supervisor domain, and is privileged to violate all access control policies. It executes with a system high security level and a system low integrity level.

The root process' main function is to respond to the IPC messages from the kernel that result from change access level requests from users at terminals. Before the root starts responding to these requests however, it carries out two activities related to system startup.

First, it carries on a dialogue with the system console to allow the modification of the system userid and password file. The system operator is allowed to add, delete, and change entries in this file each time the kernel is initialized. Once this dialogue is complete (as indicated by the operator), this file cannot be changed unless the kernel is started again.

Second, the root process spawns a system high emulator process at the system console, to allow the operator to carry out any emulator-defined system startup procedure before allowing users to log in. These procedures would typically include the setting of the system clock (which can only be set once per startup).

After spawning this emulator process, the root waits for it to die. When this occurs, the root starts accepting login IPC messages. When the root receives such an IPC message, it spawns a process at the access level requested by the user and starts the discretionary authenticator subsystem running in that process. This process is privileged with respect to violation of discretionary access controls. To pass to the authenticator the name (device identifier) of the user's terminal, the root uses IPC messages.

The above mechanism implements a multilevel terminal capability, allowing terminals to operate at any security level below some maximum. It is assumed that every user admitted to the terminal room has been externally authenticated to the maximum security level of the terminal. With this facility, the user need only inform the kernel of the security level at which he or she wishes to work.

DISCRETIONARY AUTHENTICATOR

The root spawns a discretionary authenticator each time a user requests a change of access level at a terminal. The authenticator identifies a user attempting to login and starts an emulator process to service that user's needs. Specifically, the discretionary authenticator does the following:

a. requests and saves the user's identity and password;

b. encrypts the password;

c. compares the name and encrypted password to the information in the User Data Base that contains the name, group and encrypted password of every user known to the system (This data base has security and integrity attributes such that it can be read by only the discretionary authenticator and the root, and written by only the root);

d. if the name or password is incorrect, repeats from step 1;

e. spawns an unprivileged process running the SUNIX Emulator for the user, with user and group identification that is contained in the User Data Base for this user; and

f. dies.

The discretionary authenticator executes at the access level of the terminal and is privileged to violate discretionary access controls. This privilege makes it possible to spawn an Emulator process with user and group different from that of the spawning process.

PORT MANAGER

The port manager is also started by the kernel at system initialization. The role of the port manager is to allow for interprocess communication where the messages are longer than those provided by the kernel IPC facility, and where inter-process family communication, as in the case of the Emulator, is involved. One convenient application for this subsystem is line printer spooling, where a (system high) process needs a queuing capability for data at various security levels.

The port manager executes at system high security level and at system low integrity level, and is privileged to violate the simple integrity condition and the security *-property, for it may send

messages to (modify) processes at higher integrity levels and lower security levels.

For queuing purposes, the port manager maintains a port data base. This data base consists of an array of port definitions for a set of ports defined at the compilation time of the port manager. Each port definition contains the following attributes:

a. the name of the port - an integer ranging from zero to the number of defined ports of the system;

b. the access level of the port;

c. a list of waiting messages sent to the port but as yet unreceived as well as the sending process of each message; and

d. a list of processes requesting messages through the port.

The port manager performs two operations:

a. send a message to a port; and

b. receive a message from a port.

A process desiring to send a message through a port sends an IPC message to the port manager containing the name of the port and the unique identifier (uid) of a 512-byte segment containing the message. The port manager then copies the message from the segment to a new segment at the access level of the port. The uid of the copied segment is then entered on the list of waiting messages for the indicated port. The port manager will only honor a request to transmit a message if and only if the access level of the port is greater than or equal to the access level of the requesting process and of the segment passed as a message. Whether the request is honored or not, the port manager deletes the segment containing the original message.

A process desiring to receive a message from the port sends an IPC message to the port manager containing the name of the port of interest. The port manager only honors requests to receive messages from a port from processes whose access levels are the same as the access level of the port. The port manager returns the first waiting message on the port to the requesting process. This message contains the unique identifier of the segment containing the message and the process identifier of the sending process. If no messages are waiting on the port, the port manager queues the name of the requesting process and makes the above response as soon as a message

is sent to the specified port. If more than one process is waiting for a message over a particular port, the port manager distributes incoming messages, one at a time, to waiting processes in First-In-First-Out order. The segment containing the message must be deleted by the process receiving its unique identifier.

SECTION V

THE SECURE UNIX EMULATOR ARCHITECTURE

This section describes the SUNIX Emulator architecture, and includes the structure of process families and of the Emulator data base.

The primary consideration in designing the SUNIX Emulator was to maintain maximum compatibility for existing UNIX software. In all cases where inherent protection violations existed (for instance, under UNIX, any user may read any area of core memory), compatibility gave way to security.

PROTECTION POLICY

As discussed in a previous section, UNIX protection is implemented in terms of the notions of "user", or owner, "group", and "other". The objects in the Emulator file system, directories and files, all have owner and group attributes, and a "mode" describing the read, write, and execute access granted to processes having that owner or that group or neither: the discretionary permissions. The kernel requires the additional dimension of access level, that is, security level and integrity level, for these objects. Naturally, this access level applies to every segment composing the file or directory.

In the interest of maintaining compatibility, if an access level is not expressly requested when objects are to be created by the Emulator, the access level of the process executing the command is used. If a higher access level is desired, it must be so specified.

As in UNIX, processes inherit the same user and group across a fork or execute operation. Set_user and set_group are not implemented for they allow a process to take on the attributes of another user, and access information that it might not be allowed to access under its real user/group identity. The same restriction must be made for the Superuser feature. Any implementation of the Superuser would, to some degree, allow a process to circumvent the security constraints imposed by the kernel on untrusted processes. If an Emulator process were to claim such privileges, all the Emulator software would have to be validated, violating the original concept of a small kernel.

OBJECT STRUCTURE

The Secure UNIX Emulator has two major types of objects which it maintains. These objects are:

a. process families, brought about by spawning; and

b. the file system, organized into

    a. directories,

    b. data files, and

    c. special files.

These are basically the same objects manipulated by UNIX; they differ in organization but not in function.

Information Processors: Process Family

The collection of processes generated through the Emulator by the actions of a user at a terminal is known as a process family. The first member of the family is spawned by the Discretionary Authenticator, and has the access level requested by the user for the terminal and the user and group attributes confirmed in the Authenticator's User Data Base. Subsequent members result from forks at the user interface, and execute at this initial access level.

To summarize, all process families have the following attributes:

a. a device (terminal) for standard input and output,

b. an owner,

c. a group, and

d. an access level.

The attributes of the individual processes are listed in Section III. In addition, Emulator processes have the following attributes:

a. a set of signals that may interrupt the execution of a process at any time;

b. a set of directives to control the events after an interrupt;

c. a set of open files; and

d. a local priority level.

Information Storage and I/O: File System

The Emulator file system is designed to be transparent to the user: SUNIX operations have the same effect as UNIX operations from the point of view of the user. In addition, references to directories are encouraged to be made interpretively, making it unnecessary to depend on a known directory format in implementing user software. This design policy will require that some user programs (e.g., the directory list command, "ls") be rewritten to conform to this rule; however, this is considered to be a much cleaner design.

A major change in the original file system structure is the movement of all file-related information from inodes to directories, thus making the directory entry the complete and unique definition of an Emulator defined object. This was done because of the nature of the "flat" file structure that the inode provided. UNIX inodes contain information on all the files in a given file system. Since all data is contained in kernel segments at single access levels, in order for inodes to continue to be maintained by the UNIX emulator, a separate file system would be required for each access level for which there were files, which is clearly an impossibility.

Directories

Directories provide the mapping between the names of files and the files themselves. Directories contain information about data files, special files (I/O devices), linked files, and other directories, thereby creating a tree-like hierarchy. Directories have the following attributes:

a. a pathname identifying the location of the directory in the hierarchy;

b. a collection of segments, or "pages", making up the whole directory;

c. an access level, which applies to the segments that compose it and to any data files it lists. Directories listed may have equal or higher access levels. The access level of

special files vary depending on the access level of the controlling process.

d. a time of last modification;

e. a set of locks, enabling read and write synchronization on each page; and

f. a number of entries, up to 16 per page, describing data files, links, special files, or other directories.

Every directory entry contains a file-type indicator. The amount of information in the entry varies depending on the type of file. Entries for directories contain:

a. a file name (character string);

b. discretionary permissions;

c. a unique file identifier;

d. an owner;

e. a group; and

f. the unique identifier for the first segment of the directory. Subsequent segments are doubly-linked by forward and backward pointers (unique ids on each page.)

Data Files

Emulator data files have the following attributes, all of which are recorded in their directories:

a. a character string giving the file name;

b. discretionary permissions;

c. an access level;

d. a unique file identifier: derived from the system clock to guarantee that no two files have the same identifier;

e. an owner;

f. a group;

g. a flag: indicates whether the file is small (no more than 4096 bytes) or larger;

h. a list of segments (unique ids of the segments) that make up the file. No two distinct files should contain any common segments. If the file is "large", the list identifies segments that hold the unique ids.

i. a byte count: indicates the amount of data written in the segments; and

j. an access time: the last time the file was modified.

Regular data files are made of kernel segments of fixed size (1024 bytes) and may run as large as 1364 segments (about 1.4 million bytes). This size is smaller than the maximum segment size allowable under current UNIX (16.7 million bytes), but is considered sufficient for this prototype. Data files are protected not only by their discretionary access modes (owner/group/other permissions) as in UNIX, but also by their access levels.

Links

Links, as in UNIX, are pointers to files defined elsewhere in the directory hierarchy. Links are implemented by providing in the directory entry a pathname to be used to traverse the directory tree. The link file entries in a directory contain:

a. a file name in the form of a character string;

b. discretionary permissions;

c. a unique file identifier; and

d. the pathname of the file linked to.

When a file designated as linked is referenced, the pathname is followed and file identifiers compared to obtain the directory entry of the data, directory, or special file being traced. If the destination file is deleted and another with the same name is created in the same directory, the unique file identifiers for the link entry and new entry will no longer match, resulting in an error. Note that linked files may not be linked to.

All links are performed in conformance with security constraints that preclude linking to files at lower security levels than that of the directory containing the link, and thereby creating a non-monotonically increasing directory structure.

Special Files

Special files are input/output devices defined in such a way as to make device I/O appear no different from reading and writing a data file. Special file directory entries contain the following information:

a. a file name in the form of a character string;

b. discretionary permissions;

c. a unique file identifier;

d. an owner;

e. a group;

f. a device name: a 32-bit unique integer specified at system compile time;

g. a device table index: a pointer to a list of device management routines;

h. a time: last time the device was accessed (at the indicated access level); and

i. device dependent information (e.g. "erase" and "kill" characters for terminals).

The special files correspond to the devices accessible to the user. The system disk is not considered a special file. As an initial prototype, the kernel allows only teletypes, a line printer, and magtapes to be addressed by a user. Specifically, 10 devices, 7 terminals, 1 line printer, and 2 magtape drives, will be considered special files.

Note that the access level of a device listed in the directory is the level given the device at the time the entry is created. In reality, the access level of any special file is subject to change depending on the access level of the controlling process, that is, of the user requesting service.

OPERATIONS

The Emulator operations correspond for the most part to the system calls provided by current UNIX. There are cases, however, where changes in the user interface have been effected or where

calls are no longer supported as a result of the stringent security requirements imposed by the kernel. Also, some new operations were required to make full use of the multilevel facilities. The following modifications to and incompatibilities with UNIX resulted:

a. Superuser is not supported.

b. Every terminal may have several processes executing; these processes make up a process family and communication between processes, except for IPCs, is restricted to within the family.

c. Inodes are no longer supported and data previously stored in inodes have been moved to the directories, thus drastically changing the format of a directory entry. This particularly affects the operations manipulating links (link and unlink).

d. Core memory and the kernel root file system are no longer available as special files.

e. Setuid and setgid are no longer supported.

f. Due to a completely restructured filesystem only accessed by the kernel and a desire to keep the Secure UNIX prototype small, mount commands are not implemented.

g. Access to system data such as console switches is no longer permitted.

h. The setting of the system time can be done only once after each system startup.

Additional system calls provide for the creation and deletion of files at various access levels, the moving of file entries from one directory to another, and the more exotic features of UNIX--the write and mail commands.

In all the operations described below, errors are reflected as is done in UNIX. The "c" bit in the processor status word is set and an error number is recorded in general register R0.

Process Operations

Process operations are for the most part local to the process family. Notable exceptions are operations that involve privileged

subsystems such as the port manager. Process operations fall into three categories:

a. process creation and existence;

b. intra-family communication; and

c. process status and control.

The process creation mechanism is embodied in the fork and exec operations. The fork operation, implemented with the kernel spawn, starts a new process at the same access and privilege level as the requesting process. This process recognizes the requesting process as its parent, thereby creating the family hierarchy. The exec operation makes it possible for a spawned process to execute another program.

The exit operation causes the process to end. If a successor (parent) is waiting (see below) for such an event, an IPC message is sent containing the status of the process at the time of the exit.

A wait operation allows a process to suspend execution until a descendant exits. The status of that process at the time of exit is returned via an IPC message.

Family members may signal each other by writing in the Process Family Segment mapped in each process' virtual space. Signals may be sent from one process to another through the kill operation (so-called due to the signal that causes the destination process to exit). The signal types are:

a. hangup

b. interrupt

c. quit

d. illegal instruction

e. trace trap

f. IOT instruction

g. EMT instruction

h. floating point exception

i. kill

j. bus error

k. segmentation violation

l. bad argument to a system call

With the signal operation, a process may control its actions after receiving a signal by opting either to "catch" the signal, and execute a signal handler, or ignore it. The default action on a signal causes an immediate exit. All signals, except hangup, interrupt, and quit, produce a core image if neither caught nor ignored. The kill signal alone may neither be caught nor ignored. The Emulator checks for signals at the end of each system call and acts accordingly. Note that a process may only signal other members of its family.

A number of operations allow a process to examine and/or interact with others in the family. The times operation returns the non-kernel execution times of the current process and all descendants. At the start of every Emulator system call, the virtual time spent executing in user space is recorded, and at the end, the time spent in supervisor space is marked down in a process family data base. The sum total of descendant user space and supervisor space times is returned. The process_status operation takes as argument the process id of a member of the current process family and returns the current state of the process. The process_trace operation allows a process to control the actions of a child process (direct descendant): to read and write the child's virtual space, send it signals, and control the flow of execution.

The break operation allows a user process to expand its virtual data space, providing for dynamic space allocation.

A process may control its local priority by use of the nice operation.

A process may suspend execution for a number of seconds with the sleep command.

Various operations return information about the executing process. The process_status operation may of course be used for a process to learn its own status. A process may obtain a profile of its activity through the profile operation. This operation forces records to be kept of the amount of time spent executing particular areas of user virtual memory. It is implemented through time traps, set for approximately every 1/60th of a second, that force the

program counter at the time of the trap to be examined and appropriate tallies made. The get_access_level operation returns the security and integrity level of the process performing the operation. The get_process_id operation returns the process identification number (local to the family) of the process currently executing. The get_user_id and get_group_id operations return information about the owner of the process.

File Operations

As in the case of kernel segments, processes, acting on a user's behalf, can perform three classes of operations on files: creation and deletion, modification of attributes, and read and write accesses in virtual memory.

The data file create operation takes a character string name and discretionary permissions. It creates an initial kernel segment with user and group corresponding to the user and group owning the process. The access level of the file is the same as that of the process creating it, and must correspond to the access level of the directory in which the file is to be catalogued.

The directory file create operation may take one of two forms, depending on the access level of the new directory. In either case, the character string name and discretionary permissions are required. If the directory is to have a higher access level than the process creating it, the new access level must be provided, and must be greater than or equal to the access level of the process and of the file's directory.

The change_directory operation sets the process' notion of the working directory. File names used in subsequent operations may be taken as either entries in or descendants from this directory.

To create a special file for a device, likewise a name and discretionary permissions are required. An additional argument is a pointer into a device table, which lists the Emulator device drivers for each device. The access level of the special file at the time of creation must be the same as that of its directory. The actual access level is subject to change depending on the process driving the device.

All files may be deleted with the remove_file command. If the file is a directory, the directory is recursively searched and each file and directory listed is also removed.

The link_file operation creates a link to a file, which may be a regular data file, directory, or special file. The unlink_file operation removes the link.

The move operation creates a duplicate entry for a data or special file, and removes the original file entry, in effect renaming the file. It has the effect of creating a new file, copying all the data from the old file to the new one, changing the owner, group, and discretionary permission to correspond to that of the old file, and deleting the old file.

The change_owner and change_mode operations may only be executed by the owner of a given file. They both result in modification of the meaning of the discretionary access permissions. It is important to note that, as with the segments composing a file, the mandatory access level remains fixed during the life of the file.

The operation open_file gives a process access to read and/or write the segments comprising a given file (or the device in the case of special files), and returns a "file descriptor", an integer from 0 to 14, to be used in subsequent operations on the opened file. A process may have up to 15 files opened at a given time. Mandatory and discretionary access permission to the file are applied before any file is successfully opened. In addition to the necessary discretionary access permissions, to open a file for reading, the process must have an access level greater than or equal to the level of the file (observe access); for writing, the access level must be equal to the access level of the file (modify access).

The close_file operation is the complement of open_file. This operation is particularly useful when a process requires more than the maximum number of open files during its lifetime. Any open files are automatically closed on exit.

The dup_descriptor operation allows an already opened file to be referenced by a synonymous file descriptor returned by the call.

The read_file and write_file operations allow a process to transfer data into and out of an open file. They both require a valid file descriptor, a buffer address in user space, and a byte count.

The seek operation allows positioning the byte pointer of a file at random. Seeks on pipes are forbidden, and seeks on terminal devices are meaningless.

Two operations are available to monitor and control terminal I/O. The set_tty operation allows a process to set such information as echoplexing, carriage-return/line feed conversions, baud rate, tabs, etc., on its controlling terminal. These state changes go into effect for all processes in the process family. The get_tty operation returns the results of the last set_tty. The default teletype status is that of an ASR-33.

The pipe operation creates an interprocess communication link in the form of a circular file that can be both read and written. Two descriptors are returned, one for the read end for the pipe, the second for the write end. Two processes (necessarily in the same family) with the same file descriptors for a pipe (i.e., descendant from the same process) can pass information back and forth by controlling the contents of the file.

The sync operation assures that all information placed in a file (in core) is fed out to disk.

Various status check functions are available. The stat_file takes a file name and returns the contents of the directory entry on that file. The directory_status operation takes a directory name and offset and returns the contents of the corresponding directory entry. In this case, the names of the files listed in a directory need not be known in advance. The open_file_status operation applies only to open files, and takes a file descriptor as argument.

The get_time operation returns the system's idea of the current real time.

The set_time operation sets the system clock (maintained by the kernel) the first time it is called. Subsequent calls will return no error, but will not change the clock value.

The port_manager is accessed via the port_send and port_receive operations. Both require a pointer to a buffer where up to 512 bytes of data may be stored. In the port_send operation, the data will be copied (by the Emulator) into a 512-byte segment and the uid of this segment is passed to the port manager via an IPC send request. In the port_receive operation, whatever data is waiting will be returned after the Emulator performs an ipc_rcv operation.

The inter_console_write operation allows a process to write on another user's console. It requires the user id of a person presumably logged in and the address of a buffer containing the data to be written. The terminal for the receiver is determined and the data is written via an iowrite kernel operation. Service may be denied if the access level of the receiver terminal is less than the

access level of the sender, or if discretionary permission to the terminal prohibits access.

EMULATOR INTERNAL ARCHITECTURE

The internal architecture of the Emulator in many ways resembles that of current UNIX. User system calls are vectored through a trap handler that makes the appropriate function call. During the course of the operation, various process family data bases are accessed, and care must be taken to coordinate the read/write operations with those of other processes. A device may be accessed or a signal received and processed. And throughout, a process is responsible for scheduling itself, that is, relinquishing the CPU if it determines it has used more time (since its last DOZE or RECEIVE) than is equitable for the balanced operation of other processes.

Data Bases

Data bases available to the Emulator fall into two classes:

a. process family related, or

b. process local.

Of those associated with a process family, they may fall into one of three categories:

a. data file related;

b. special file (I/O device) related; or

c. process status related.

Process local information ranges from pointers to system call arguments to timing variables that must be preserved across system calls.

Process Family

All process family data is collocated in a Process Family Segment shared by all members of the family. This segment holds three tables:

a. the process table;

b. the active file table; and

c. the active object table.

For each of these tables, a lock mechanism is provided to synchronize read and write accesses.

The process table holds the information on each process that allows family members to communicate among themselves:

a. the current status of the process, if recently forked, running, waiting to die, waiting for a child to die, being traced;

b. the family local ids for this process and its parent;

c. the kernel id for the process;

d. the last signal received;

e. the uid of the Emulator stack for this process; and

f. if the process is in a wait state, the event for which it is waiting.

The active object structure contains one entry for every file, pipe, and directory currently in use by any process in the family. Information for the object referenced is copied into these entries from the directory entry:

a. the file type: directory, special file, or data file, and if data, large or small;

b. discretionary permissions;

c. the unique file id of the object;

d. the owner of the file;

e. the group;

f. uids of the basic segments that make up the file;

g. the size of the file in bytes;

h. the unique id of the directory page that catalogues the file; and

i. the number of active file table references to this entry (e.g., if a file is opened by more than one process).

The active file table holds an entry for every open operation. The items in each entry are:

a. a flag indicating if the file is opened for reading, writing, or both and if the file is a pipe;

b. a reference count of the number of duplicates on this entry (e.g. a result of a dup_descriptor operation);

c. the address of the active object table entry for the file; and

d. the offset in the file for the next read or write.

The addresses of all device drivers addresses are found in a device configuration table along with other pertinent data:

a. the device identifier;

b. a flag indicating if device is a terminal; and

c. addresses for the open, close, read, and write functions.

Process Local

Each process has on its stack a number of static variables that need to be maintained across system calls but are not necessarily of interest to other family members, and data that must be maintained across function calls while the Emulator is executing. These items include:

a. the user's arguments to the last system call;

b. the address of the user's hardware (general purpose) registers;

c. signal direction indicators;

d. an error type (may be null);

e. the address of the name (character string) of the last file referenced;

f. owner, group, and access level of the process;

g. the addresses of functions to be executed in case of interrupts or time traps;

h. profiling flag and arguments;

i. the current process' user and system times and the sum of its children's user and system times;

j. a list of active file table entries for files open to this process;

k. the unique id of the process family segment;

l. the unique id of a directory at a higher access level being created or deleted;

m. the uids and justifications of segments residing in user domain pages;

n. the uid of the last segment stored in the Emulator's utility register;

o. the address of the active object table entry of the process' working directory;

p. the current address in user space for the next byte transfer;

q. the current address in a file for the next byte transfer;

r. the number of bytes remaining for transfer into or out of a file;

s. the address of the process table entry for this process;

t. trap flag and caught traps;

u. access privileges;

v. text, data and stack sizes of user code; and

w. scheduling variables.

Emulator Subsystems

The SUNIX Emulator contains special management routines for internal process control. These routines allow for the following operations:

a. trap management;

b. signal transmission and processing;

c. scheduling;

d. locks on data bases; and

e. device management.

Trap Manager

Traps are processed by the Emulator's trap management routines. This manager is responsible for calculating the user and supervisor times that are sent to the user during a child_times operation. Its primary task is to react to traps initiated either by a user or as the result of an interprocess trap operation through the kernel. The trap manager processes the following trap types:

a. bus error;

b. illegal instruction;

c. breakpoint trap;

d. I/O trap (IOT instruction);

e. emulator trap (EMT instruction);

f. floating exceptions;

g. segmentation exception (MMU faults);

h. time traps;

i. system calls;

j. terminal I/O traps;

k. non-terminal I/O traps;

l. interrupts;

m. quits; and

n. hangups.

The first seven trap types result in signals being sent to the process. Traps from system calls cause the arguments to be placed in the Emulator's U vector and the desired function to be called. At the end of every trap, the signal manager is invoked.

Signal Manager

A signal has no direct reaction on a process. It merely sets a flag (in the process table) that asks a process to do something to itself.

When a process detects a signal, it may or may not enter a "STOP" state, depending on whether or not it is being traced. In the STOP state, it will wait for a message from the tracing process to tell it what to do, which may be one of the following:

a. return the contents of a user location in I or D space;

b. change the contents of a user location in I or D space;

c. return the contents of the U vector;

d. change the contents of the U vector;

e. process a given signal;

f. exit.

All communication is done through IPC messages.

One process may debug a program executed by another process by requesting a breakpoint be written at a strategic location in instruction space; the resultant breakpoint trap (on execution at that address) would force the tracee to signal itself and then standby for further requests (e.g., examine registers, local variables, etc.).

If the process is not being traced, the corresponding signal table entry is referred to for the response:

a. Zero: default action; in the case of certain signals, create a "core" file in the current directory (access permitting) and copy the U vector and user virtual space into it, along with hardware registers; then exit.

b. Any odd number: ignore the signal and continue.

c. Any even number: treat the entry as an address and return to it in user space.

Wait Manager (Scheduler)

An Emulator process invokes the scheduler preceding all device I/O or when it determines it has used a predetermined amount of uninterrupted user and supervisor time. If the scheduler determines a time-out condition, it DOZEs to allow higher priority processes to continue. If the process is doing an extraordinary amount of I/O, the scheduler simply resets its priority to something relatively low.

Lock Manager

Locks are used to coordinate reads and writes on data bases accessed by members of a process family, and in certain cases processes outside of a family. Locks are effective only if processes cooperate in the locking strategy. This mechanism is handled by the lock management routines:

a. lock for reading;

b. lock for writing;

c. unlock after reading;

d. unlock after writing.

These routines all operate on a lock structure for each data base containing:

a. a lock lock: used to guarantee exclusive access to the lock structure. This is particularly critical in the case of multiprocessors.

b. a modify count: incremented during a write lock operation to indicate impending change to the data base;

c. a lock count: records how many processes are waiting on the data base; and

d. a process id: the unique id of the last process requiring the data base.

Read locks are only performed if non-exclusive access is sufficient. A read lock operation on a data base returns the value of the corresponding lock's modify count. If a write to the data

base (in the form of a write lock) occurs before the read unlock, the value returned by the read unlock function is non-zero. The calling routine is expected to repeat the read-lock/read/read-unlock sequence until zero is returned.

If all write accesses are preceeded by a call to the write lock mechanism, it will provide for exclusive access to a data base. A process first examines the lock lock, its only indication that another CPU needs the data. Then, by examining the lock count, a process knows if the data base has already been locked. It copies the process id field onto its stack, replacing it with its own id, and waits for a message generated by the write-lock operation. At that time, it restores the old process id, increments the modify count, and continues its write operation. When done, it executes a write-unlock, which increments the modify count and sends an IPC message to whatever process is named in the process id field. This method of queuing waiting processes results in a Last-In-First-Out ordering.

During the write locks and unlocks, all traps are deferred, that is, queued, to prevent a data base from being locked up but never released (due, for instance, to a kill signal).

Device Manager

Access to devices is transparent to user software due to the attempt in UNIX to make device I/O appear as straightforward as file read and write operations. This property is carried over in the SUNIX Emulator. If a file to be read or written happens to be a terminal, magtape, or the line printer, control is switched to the appropriate routines via the device configuration table described above.

Terminal I/O requires some pre- and post-processing due to the many options given a user for his terminal. For instance, erase and kill characters must be processed; the meanings of carriage return and line feed characters may be modified (in general to produce a combination carriage-return/linefeed or newline character sequence); delays may need to be inserted (in the form of rubout characters) after carriage returns, tabs, and form feed characters; or raw mode may be desired, in which case, the erase and kill characters lose their special meaning.

Similarly with the line printer, care must be taken that not more than, for instance, 132 characters are placed on a line (the others are not printed); again, tabs must be converted to spaces, and carriage returns and line feeds processed appropriately.

In the case of terminals, the user may determine the characteristics he wants for this terminal, through the set_tty operation. There is no such option for the line printer.

SECTION VI

SUMMARY

This section will summarize the major features of the Secure UNIX prototype and outline the disappointments and hard problems encountered.

ACCOMPLISHMENTS

The SUNIX Prototype project began with the realization that a "simple" kernel would probably not be efficient in the support of a Secure UNIX. To efficiently support a secure UNIX system, the security kernel should provide:

a. support for a large (about 100) number of processes not always core resident - meaning that the kernel must support the swapping and scheduling (without storage channels) of a significant number of processes;

b. support for DMA devices;

c. efficient (and consistent) support for character oriented terminals;

d. kernel reentrancy/concurrency to minimize overhead in the multi-process environment;

e. support for multiple sizes of segments, recognizing that different mechanisms in SUNIX (the process image, the file system, and the SUNIX Emulator) all require differing sizes of object;

f. support for fixed size (clock derived) object names asserting that this fact substantially simplifies the construction of system software in the kernel provided environment;

g. support for timing and clocking mechanisms with about one millisecond resolution;

h. generality in the kernel; and

i. support for a large number of security levels (sixteen classifications and sixty-four categories).

The combination of these factors necessitated a reexamination of the architecture of the existing security kernel for the PDP-11/45 [9].

The resulting design, documented herein, incorporates a number of state-of-the-art concepts to address, and minimize, the kernel complexity induced by the above "requirements". Some of these concepts are:

a. the specification of the kernel/non-kernel interface (objects and operations) via a formal behavioral specification;

b. the decomposition of the kernel implementation into well-defined abstract machines with a careful allocation of reponsibility to each level of machine;

c. the orthogonal decomposition of the kernel implementation into parallel processes, communicating via semaphores and messages, carefully controlling access to shared data bases so as to minimize uncertainty caused by parallelism;

d. the elimination of interrupts at a very low level within the kernel;

e. the elimination of traps within the kernel, e.g., the kernel never takes an address fault;

f. the decomposition of the notion of process into four levels: hardware, primitive, eligible, and user - hopefully clarifying an often ambiguous area of system design; and

g. the design of an effective mechanism for the preemption of a segmented virtual memory based on the "buddy" system of memory allocation.

The incorporation of these mechanisms does not impose an intolerable verification burden. Verification is complicated by (at least) two issues: size and complexity. There is no escaping the conclusion that the design, as presented, is more complex than earlier kernel designs [9]. However, the design offers better facilities and substantially more parallel performance. The internal concurrency of the kernel will cause verification problems, yet its architecture seems to prevent the uncertainty of state that interrupt-like mechanisms cause. The provision of a well-defined concurrency does not appear to preclude verification, even at the current state-of-the-art.

DISAPPOINTMENTS

Despite the above discussion, the largest disappointment of the design is the size and complexity of the resulting system. While (seemingly) many times better than previous small operating systems for PDP-11s, its size and complexity seem just at the limit of comprehension for a single designer or architect. Until its performance (in terms of utility, performance, and ease of verification) have been demonstrated, the success of the SUNIX Prototype will not be demonstrated.

The SUNIX Emulator, in general, seems to be quite well constructed. The very notion of partitioning the system into kernel and emulator appears to be architecturally sound. The system appears rather elegant. However, several problems do occur. The major ones relate to the communication and synchronization of different families of processes. The two major instances are:

a. the control of one process family by another - for instance, there appears to be no feasible means (barring privileged process intervention) to kill a process in one process family from another process family - families are just too independent. This consideration has related concerns in other areas of pragmatic system management, viz., identification of all existing but unused segments, the reconstruction of broken filesystems, secure backup/retrieval (staging) of information to bulk backup store, etc., and

b. the coordination of access to files shared by processes in differing process families - there exist conceptual solutions to this problem but they are unattractive due to increased disk access (viz., placing a lock on the file (in its directory entry) that must be accessed for each file access - this solution requires access to the directory for each access to the file - this might be alleviated somewhat by a new access mode - write with lock - that will operate in the correct manner but only if specifically requested by the using process).

Another disappointment is the removal from the design of several kernel operations - having nothing to do with security yet of possibly central importance in a working, operational system. These two kernel operations were intended to facilitate a "survey" of a kernel filesystem, and were not fully designed and implemented in the interest of time.

These operations would, for a given unprivileged process, provide the mechanism by which that process can ascertain the names of all segments to which the process has access. In this manner, a system high process, privileged to violate discretionary protection, could find any inconsistencies in the Emulator supported filesystem. That is, "lost" segments that exist of which the Emulator has no record, and "missing" segments that the Emulator believes to exist yet of which the Kernel has no knowledge, could be found. This mechanism took the form of two kernel operations:

a. reset_filesystem: this kernel operation specifies to the kernel the name of the filesystem to be surveyed; and

b. next_segment: this kernel operation returns the name (uid) of the "next" segment stored on the filesystem - no guarantee of sequencing is made only that each existing segment will be reported once, concurrent creation of new segments may result in new segments not being reported.

HARD PROBLEMS

A major problem with secure system design is the construction of large objects (such as user files) out of small objects (such as the kernel-supplied segments), outside of the security sensitive software. Since secure environments provided by security kernels are by definition partitioned rather formally with respect to access level, it is difficult to design effective mechanisms for the construction of objects (available to a number of different processes at differing access levels) from smaller objects. It appears to have no easy answer short of making all software responsible for the construction of objects security sensitive.

This problem is the basic root cause of both of the Emulator problems mentioned above (interprocess family kill and file access coordination). The Emulator problems could be eased by having a single Emulator-managed process table and having information about all processes in it. However, since the information in the table is about processes at different levels, only a subset of the table could be accessed by each process. Clearly, if such a table did exist, then it would have to be maintained by the kernel (or other security sensitive software).

Another problem can be attributed to the hardware, particularly the memory management hardware. The problem is the overhead associated with the switching of address space registers for each process swap. A primary motivation in the placement of terminal device drivers within the kernel domain was the avoidance of a full

address space switch for every character input. These considerations motivated the design of four levels of process: hardware, primitive, eligible, and user. A Multics design concurrently and independently discovered and implemented, only requires three levels of process due to hardware support that allows more flexible representation of address spaces (placement in memory, with processor support for a pointer-to-memory switch).

REFERENCES

1. Ritchie, Dennis M. and Ken Thompson,"The UNIX Time-Sharing System," Communications of the ACM, Volume 17, Number 7, July 1974, pp. 365-375.

2. Thompson, K., and Ritchie, D. M., UNIX PROGRAMMER'S MANUAL, Bell Laboratories, Murray Hill, N. J., May 1975.

3. Department Of Defense, "Security Requirements for Automatic Data Processing (ADP) Systems," Department of Defense Manual 5200.28, December 1972.

4. Department of the Air Force, "Security Requirements for Automatic Data Processing Systems (ADPS)," Air Force Regulation 300-8, June 1974.

5. Bell, D. E., L. J. LaPadula, "Secure Computer Systems," ESD-TR-73-278, Volumes I-III, Electronic Systems Division, AFSC, Hanscom AFB, MA, November 1973, November 1973, April 1974, AD 770768, AD 771543, AD 780528.

6. Biba, K. J., "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, Electronic Systems Division, AFSC, Hanscom AFB, MA, April 1977, AD A039324.

7. Woodward, J. P. L., "Design and Abstract Specification of a Multics Security Kernel," ESD-TR-77-259, Vol. III, Electronic Systems Division, AFSC, Hanscom AFB, MA, March 1978, AD A053149.

8. Knuth, Donald E., The Art of Computer Programming, Volume 1, Addison Wesley Publishing Company, Reading, Massachusetts, pp. 442-460.

9. Schiller, W. L., "Design and Specification of a Security Kernel for the PDP-1145," ESD-TR-75-69, Electronic Systems Division, AFSC, Hanscom AFB, MA, May 1975, AD A011712.